TAMPEREEN TEKNILLINEN YLIOPISTO

NINA ANTTILA

SULAVA SIIRTYMÄ 3D- JA 2D-OBJEKTIEN VÄLILLÄ

Diplomityö

Tarkastaja: professori Timo D. Hämäläinen
Tarkastaja ja aihe hyväksytty Tieto- ja sähkötekniikan tiedekuntaneuvoston kokouksessa 27. syyskuuta 2017

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
ANTTILA, NINA: Sulava siirtymä 3D- ja 2D-objektien välillä
Diplomityö, 46 sivua
toukokuu 2018
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Timo D. Hämäläinen
Avainsanat: ohjelmistovisualisointi, 3D-visualisointi, tietokonegrafiikka, Qt 3D

Ohjemistojen visualisointi on tärkeä osa ohjelmistokehitystä, sillä graafisella kuvalla voidaan välittää informaatiota järjestelmästä sanallista dokumentaatiota paremmin. Ohjelmistot ovat kuitenkin rakenteiltaan monimutkaisia ja kaiken tarvittavan informaation tehokas välitys voi olla hankalaa, sillä perinteiset kaksiulotteiset visualisointimenetelmät, kuten graafit ja listat, muuttuvat sekaviksi informaation määrän kasvaessa. Apua ongelmaan pyritään löytämään kolmiulotteisista visualisointitekniikoista, joilla voidaan kiteyttää suurempi määrä informaatiota kuviin.

3D tuo kuitenkin mukanaan haasteita. Navigointi näkymässä on haastavampaa vapaamman liikkuvuuden takia ja pahimmillaan käyttäjä voi eksyä näkymään. Perinteisiin käyttöliittymiin tottuneen käyttäjän voi olla myös hankala hahmottaa, että 3D-objektit sisältävät toimintoja. 3D ei myöskään sovellu kaikkeen, vaan osa informaatiosta on edelleen suotavaa esittää kaksiulotteisin menetelmin. Lisäksi myös 3D:lläkin tulee raja vastaan informaation määrän kanssa.

Ohjelmistokokonaisuuden visualisoinnissa on hyvä yhdistää sekä 3D- että 2D-visualisointitapoja. Tässä työssä selvitetään miten siirtymä eri visualisointimenetelmien välillä saadaan sulavaksi toteuttamalla prototyyppi ohjelmasta, jolla mallinnetaan ohjelman hierarkiarakenteita 3D-objekteilla ja komponenttien yksityiskohtia, muokattavia tekstidokumentteja, 2D-objekteilla. Prototyyppi toteutetaan Qt:lla ja sen 3D-piirron moduulilla Qt 3D:llä.

Siirtymä 3D- ja 2D-objektien välillä saadaan sulavaksi käyttämällä pehmeää liikettä, navigaation rajaamista sekä maamerkkejä ehkäisemään käyttäjän eksymistä. 3D:stä 2D-näkymään ja hierarkiatasojen väliseen siirtymiseen käytetään semanttista zoomausta, jolla hallitaan myös näkymässä kerralla näytettävää tietoa. Interaktiota näkymän kanssa korostetaan animaatioilla.

Prototyyppi toteutettiin ensisijaisesti korvamaan perinteiset 2D-valikot Kactus2-ohjelmassa, joita käytetään kuvaamaan IP-XACT-kirjastojen hierarkiarakennetta. Kactus2:n ohella prototyypin idea on myös sovellettavissa muihin ohjelmistosuunnittelussa käytettäviin järjestelmiin esimerkiksi kuvaamaan projektin kansiorakenteita tai eri moduulien suhteita toisiinsa.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
ANTTILA, NINA: Smooth transition between 3D and 2D objects
Master of Science Thesis, 46 pages
May 2018
Major: Software engineering
Examiner: Professor Timo D. Hämäläinen
Keywords: software visualization, 3D visualization, computer graphics, Qt 3D

Visualizing a software system is an important aspect of development as a graphig image can convey information about the system with much higher effectiveness than a textual description. However, complex structures can make it difficult to convey all necessary information since traditional methods, such as graphs and lists, can become chaotic when more information is added to them. A solution to this problem might be found by using 3D visualization techniques which allows greater information density within the image.

However, 3D can make implementation challenging. Navigating within the scene can be confusing due to higher degree of freedom and in the worst case scenario the user can become disoriented. A user accustomed to traditional widget-based user interfaces can find it difficult to do actions with 3D objects. Nevertheless, 3D does not fit to represent all types of information. Sometimes 2D representation can do the job better. In addition, even 3D has its limitation with the information load.

To visualize a whole software structure, both 3D and 2D methods are used. In this thesis the transition between 3D and 2D objects and how it is done smoothly are investigated by implementing a prototype in which hierarchical component structures are visualized in 3D and their details, editable text documents, are represented as 2D objects. The prototype is done by using Qt and its rendering module Qt 3D.

The smooth transition between 3D and 2D objects is achieved by using fluid motion, navigation restrictions and landmarks to prevent user disorientation. Transition from 3D to 2D as well as between hierarchy levels is done with semantic zoom which is also used to control which information is shown at the time. Interaction with the scene is indicated with the use of animations.

The prototype was designed to be used with Kactus2 to replace traditional 2D menus which are used to visualize hierarchical structures of IP-XACT libraries. In addition to Kactus2, the idea behind the prototype can be applied to other software tools to be used in visualizing folder structures or component relations.

ALKUSANAT

Tämä diplomityö tehtiin Tampereen Teknillisen Yliopiston tietotekniikan laboratoriolle.

Haluan kiittää professori Timo. D. Hämäläistä työn ohjauksesta. Lisäksi haluan erityisesti kiittää tohtorikoulutettava Esko Pekkarista työn toteutuksen sekä kirjoitusprosessin aikana saadusta tuesta ja palautteesta.

Tampereella 17.5.2018

Nina Anttila

SISÄLLYS

KUVALUETTELO

TERMIT JA NIIDEN MÄÄRITELMÄT

Kactus2	IP-XACT-pohjaisten järjestelmien suunnittelutyökalu
Kehysgraafi	Qt 3D:ssä käytetty OpenGL-piirtoliukuhihnan vaiheiden konfiguraattori
Maisemagraafi	Qt 3D:ssä käytetty 3D-näkymän konfiguraattori
Piirtotekniikka	ks. kehysgraafi
Qt 3D	Qt-kehitysympäristön piirto- ja simulaatiomoduuli
VLNV	IP-XACT-komponentin tunniste

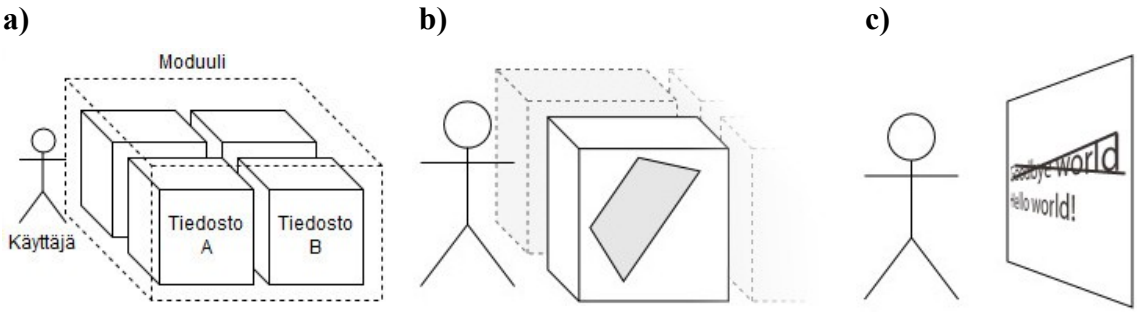
1 JOHDANTO

Ohjelmistokehittäjät käyttävät paljon aikaa olemassa olevien ja uusien järjestelmien ymmärtämiseen. Ohjelmistokehitystä varten on kehitetty useita visualisointityökaluja ja -tekniikoita [1], joilla pyritään selventämään ohjelmien rakenteita ja toiminnallisuutta. Perinteisesti työkaluilla luodaan ohjelmasta kaksiulotteinen kuva, jossa ohjelman komponentit ja niiden väliset suhteet on kuvattu erilaisin graafein, puin sekä listoin. Graafisilla kuvilla on tärkeä rooli ohjelmistokehityksessä, sillä ne auttavat hahmottamaan suuria ja monimutkaisia kokonaisuuksia paremmin kuin tekstimuotoiset dokumentit.

Ohjelmakoon kasvaessa, visualisointikuvissa olevan tiedon määrä lisääntyy, eivätkä graafit tai listat enää riitä tehokkaaseen tiedon välittämiseen ja omaksumiseen. Tähän on pyritty löytämään apua kolmiulotteisista visualisointimenetelmistä [2]. Kolmatta ulottuvuutta voidaan käyttää lisätiedon esittämiseen tai sitä voidaan hyödyntää sen viihteellisyyden ansiosta, mutta se soveltuu eri tehtäviin kuin kaksiulotteiset menetelmät: 3D toimii hyvin kokonaiskuvan hahmotuksessa, mutta tarkemmissa yksityiskohdissa perinteiset 2D-menetelmät toimivat paremmin.

Tämän työn motivaationa toimii suunnitelma SoC-kehityksen pelisöinnistä [3]. Sen tarkoituksena on tuoda pelien viihteellisyys osaksi suunnitteluprosessia mallintamalla suunnitelma 3D-maailmana ja lisäämällä kehittäjä maailmaan pelaajana. Lisäksi sen avulla pyritään luomaan yhteys visualisoinnin ja varsinaisen kehityksen välille: visualisointia käytetään yleensä uusien järjestelmien suunnitteluun ja vanhojen dokumentointiin sekä analysointiin, mutta kehitysaikana tehdyt muutokset eivät päivity kuviin itsekseen. Kun visualisointi ja muokkaus yhdistetään pelimaailmassa, mahdollisesti nopeutetaan kehittäjän tutustumista suunnitelmaan ja kehitystyön tekemistä.

Tässä työssä haluttiinkin selvittää, kuinka siirtymä eri visualisointitapojen välillä saadaan sulavaksi. Työssä halutaan toteuttaa prototyyppi ohjelmasta, jolla voi siirtyä 3D-näkymästä 2D-näkymään ja takaisin. Siirtymää on havainnollistettu kuvassa 1. Kaksiulotteinen näkymä halutaan toteuttaa siten, että siinä esitettävä tieto on muokattavissa. Kaksiulotteinen näkymä rajataan tämän työn yhteydessä tekstimuotoiseen dokumenttiin.



Kuva 1: a) Käyttäjä tutkii suunnittelemaansa moduulia 3D:nä. b) Käyttäjä päättää muokata tiedoston A sisältöä. c) Käyttäjä siirtyy 2D-näkymään tekemään muokkauksen.

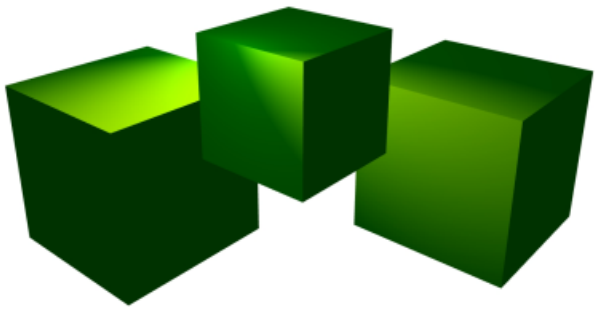
Koska työssä kehitetään prototyyppiä ja haetaan vasta tuntumaa siiheen, miten varsinainen järjestelmä toteutetaan, ei toteutuksen yhteydessä ole käytetty erillistä testiryhmää. Sen sijaan toteutus pohjautuu eri toteutustapojen kokeiluun ja kehittäjän, sekä muita prototyypistä palautettta antaneiden tuntumaan siitä mikä tuntuu luontevalta.

1.1 Olemassa oleva prototyyppi

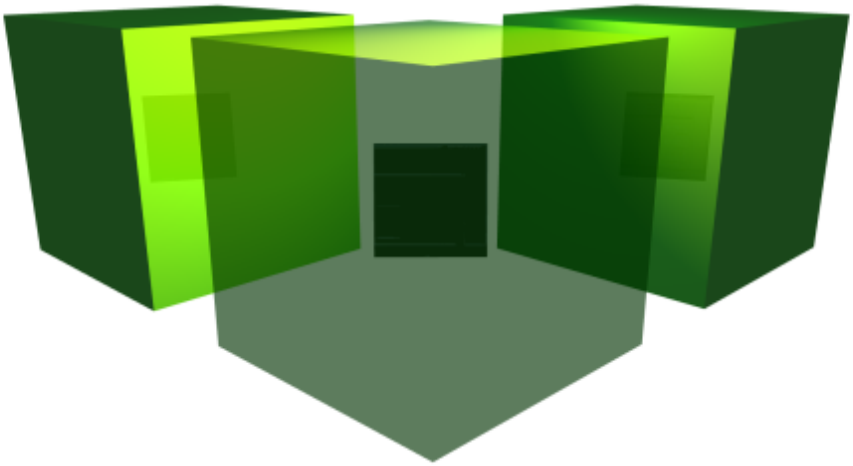
Tämä diplomityö lähdettiin tekemään Qt 3D:lla tehdyn prototyyppirungon päälle. Prototyypissä erilliset komponentin osat, yksittäiset tiedostot, mallinnetaan arkkitehtuurin tasolla kolmiulotteisina laatikoina. Laatikot reagoivat kameran sijaintiin muuttumalla sitä läpinäkyvimmiksi, mitä lähemmäksi kamera siirtyy ja vastaavasti väri muuttuu kiinteämmäksi kameran siirtyessä etäämmälle.

Yksi työn vaatimuksista on korjata piirron ongelma, jota havainnollistetaan kuvassa 2. Läpinäkyvyyden käyttö aiheuttaa laatikoiden piirtymisen väärässä järjestyksessä toistensa päälle tietyistä kuvakulmista katsottuna. Ongelman ratkaisua käsitellään luvussa 4.2.1.

Kunkin laatikon sisällä on taso, jonka materiaalin tekstuurina käytetään kuvakaappausta oikeasta tekstieditorista. Näillä simuloitiin varsinaista tekstieditorin näkymää, joka toteutettiin osana työtä. Editorin toteutusta käsitellään luvussa 5.3. Sisänäkymien



Kuva 2: Läpinäkyvyydestä aiheutuva renderöinnin ongelma näkyy tietyistä kuvakulmista. Keskimmäisen laatikon pitäisi piirtyä muiden taakse.



Kuva 3: Sisänäkymä paljastuu, kun kamera siirtyy lähemmäksi laatikkoa.

toimintaa havainnollistetaan kuvassa 3. Ne paljastuvat, kun laatikot haalistuvat kameran lähestyessä ja kääntyvät katsomaan kameraa kohti kameran sijainnin muuttuessa.

Kameran liikkeistä vastaa Qt 3D:n oma QOrbitCameraController, jonka avulla pystyy kaartamaan katselukohteen ympärillä. Tässä työssä ohjain korvattiin omalla ohjaimella, koska QOrbitCameraController ei ollut työn kannalta mielekäs ohjauskomentojen ja liikkeiden toteutusten kannalta. Perusliikeiden ohella tarvittiin ohjauksen erottelu laatikoiden sisä- ja ulkopuolella sekä kommunikointia muiden objektien välillä. Ohjausta käsitellään luvussa 5.1.

1.2 Vastaavat toteutukset

Siirtyminen kaksi- ja kolmiulotteisten näkymien välillä ei ole uusi idea. Esimerkiksi 3D-pelinäkymään liitettyjä käyttöliittymäelementtejä voi löytää perinteisten videopelien ohella VR- ja AR-sovelluksista ja moderneista pelimoottoreista, kuten Unitysta ja Unreal Enginestä, joista löytyy valmiita komponentteja tähän tarkoitukseen [4][5]. Ainakin Unityn ratkaisu perustuu widgetin piirtämiseen tekstuurina monikulmioverkon pinnalle.

Dead Space -pelisarjassa puolestaan peliin upotettua Tram Systemin käyttöliittymää käytetään siirtymänäkymänä kenttien välillä [6]. Siirtymän alkaessa kamera zoomaa pelinäkymästä käyttöliittymän sisään ja näkymä toimii latauskuvana. Siirron lopuksi kamera zoomaa ulos näkymästä takaisin pelimaailmaan. Toteutukseltaan käyttöliittymä perustuu hiukkasjärjestelmään tekstuurin sijasta.

Pelien tapauksissa suurin ero työhön on se, että niiden sisältö on pitkälti ennaltamäärätty. Sisällöllä harvemmin on mitään tekemistä käyttäjän omien tiedostojen

kanssa, lukuunottamatta käyttäjän kiintolevyllä sijaitsevia pelin omia tiedostoja. Läheisemmin työn keskeistä ideaa muistuttaa WolfenQt [7], joka on myös toteutettu Qt:lla. Ohjelmassa Qt:n käyttöliittymäwidgetit piirretään pelinäkymän käytävien seinille ja widgetit ovat käytettävissä kuin perinteisessä käyttöliittymässä. Käyttäjä voi mm. katsella omia videoitaan niillä. WolfenQt:n toteutuksessa ei kuitenkaan ole käytetty Qt 3D:tä ja sen toteutus perustuu QGraphicsItemien projisoimiseen näkymän seinille.

Sen sijaan ohjelmistovisualisoinnin yhteydessä työn kaltaista siirtymää 3D- ja muokattavan 2D-näkymän välillä ei tiettävästi ole ennen tehty. Sen lisäksi, että 3D-näkymässä voi katsella 2D-objekteja, on otettava huomioon visualisoinnin yhteydessä esiintyviä ongelmakohtia, kuten navigoinnin ja sisällönrajauksen toteutusta. Näitä ongelmia käsitellään luvussa 2.

1.3 Työn rakenne

Luvussa 2 tutustutaan työn kannalta oleellisiin tekijöihin visualisoinnin kannalta. Visualisoinnin yhteydessä esiintyy monia potentiaalisia ongelmia, jotka tulee ottaa huomioon näkymässä esimerkiksi navigoinnin yhteydessä tai kun mietitään kuvan yhteydessä esitettävää informaatiota. Luvussa käsitellään niin 3D:n käyttöä visualisoinnissa kuin käyttäjälle näytettävän sisällön hallintaa.

Luvussa 3 esitellään työssä käytetty teknologia, Qt 3D, joka on Qt-kehitysympäristön piirto- ja simulaatiomoduuli. Luvussa käsitellään Qt 3D:n arkkitehtuurin oleellisimmat tekijät: maisema- ja kehysgraafi. Lisäksi tutustutaan työn toteutuksen kannalta oleellisiin Qt3DWindowiin ja 3D-näkymän koordinaatistoon.

Luvut 4 ja 5 käsittelevät prototyypin toteutusta. Luvussa 4 perehdytään hierarkiarakenteen pohjalta rakennettavan 3D-näkymän luontiin ja siihen, miten 2D-objektit esitetään sen yhteydessä. Luvussa 5 käsitellään navigaatiota, käyttäjän interaktiota maiseman objektien kanssa sekä tekstieditoria.

Luvussa 6 käsitellään työn arviointia. Arvioinnissa pohditaan niin työn tavoitteiden toteutumista kuin Qt 3D:n soveltuvuutta työhön ja mahdolliseen jatkokehitykseen. Lisäksi luvussa käsitellään jatkokehitystä prototyypin parantamisen ohelta sekä integrointia osaksi Kactus2:ta.

Luku 7 sisältää yhteenvedon ja lopulliset johtopäätökset.

2 VISUALISOINTI OSANA OHJELMISTOKEHITYSTÄ

Visualisoinnilla on tärkeä rooli ohjelmistokehityksessä, sillä se auttaa ymmärtämään uusia ja olemassa olevia järjestelmiä. Visualisointia varten on kehitetty useita erilaisia työkaluja, joilla voidaan havainnollistaa ohjelmistojen rakenteita komponentti- ja lähdekoodin tasolla, ohjelman osien suhteita toisiinsa, ajoaikaista suoritusta sekä järjestelmän kehitystä ajan saatossa [2][3]. Perinteiset visualisointimenetelmät ovat yleensä pohjautuneet kaksiulotteisiin kuviin ja kaavioihin, mutta nykyään myös kolmiulotteiset visualisointitavat ovat yleistymässä.

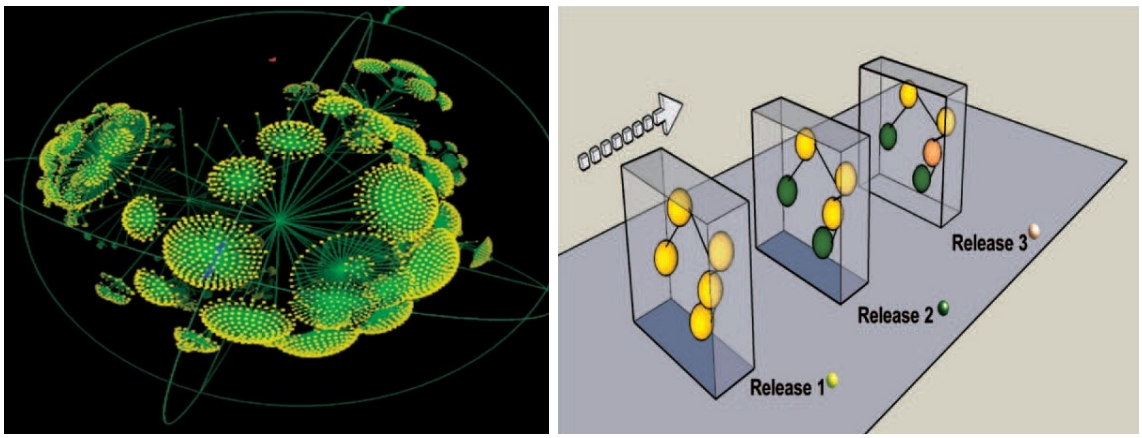
2.1 3D osana visualisointia

3D on ihmiselle luonnollinen ja helposti omaksuttava tapa kuvata asioita ja siihen näköjärjestelmämmekin on sopeutunut – elämmehän maailmassa, joka koostuu kolmiulotteisista objekteista. Kolmiulotteinen esitystapa tekee kuvasta viehättävämmän, mutta myös intuitiivisemman ja helpommin muistettavan. Yksinkertaisimmillaan sitä voidaankin käyttää esteettisistä ja viihteellisistä syistä, mutta siihen voidaan myös sitoa lisäinformaatiota. [2]

Eräs visualisoinnin tärkeimmistä ominaisuuksista on se, että se kokoaa suuren tietomäärän tiiviiseen, mutta ymmärrettävään muotoon. Kun ohjelmistokokonaisuus koostuu useista sisäkkäisistä rakenteista ja komponentit muodostavat eritasoisia suhteita toisiinsa, alkaa tilankäytön suhteen tulla ongelmia: kaikkea ei saa mahtumaan yhteen kuvaan siten, että kokonaiskuva pysyisi selkeänä, mutta samanaikaisesti yksityiskohtia ymmärtäisi. Tämä tulee ongelmaksi etenkin kaksiulotteisissa kaavioissa, jotka muuttuvat sekaviksi, kun kuvan yhteyteen lisätään yksityiskohtia ja lisätietoja [1][3].

Kun käyttöön otetaan kolmas ulottuvuus, saadaan lisää tilaa kolmannelta akselilta. Tilaa voi esimerksiksi hyödyntää elementtien sijoittelussa, jolloin objekteja saadaan sijoiteltua toistensa taakse. Kuvassa 4a suuren graafin esityksessä on hyödynnetty kolmatta ulottuvuutta, jolloin koko graafin saa mahtumaan pienempään tilaan. Kolmatta akselia voidaan myös käyttää parametrien mittaamiseen kahden muun akselin lisäksi. Kuvassa 4b hyödynnetään kolmatta akselia kuvaamaan järjestelmän muutosta ajan saatossa. Voidaankin sanoa, että 3D-esitystapa mahdollistaa suuremman informaation määrän kuvassa. [2]

a) **b)**



Kuva 4: a) Suurien graafien tilankäyttöä voidaan tehostaa, kun solmujen sijoittelussa hyödynnetään kolmatta ulottuvuutta [2]. b) Järjestelmämuutoksia voidaan visualisoida esittämällä järjestelmän tila kaksiulotteisena näkymänä ja muutos ajansaatossa kolmannella akselilla [2].

3D ei kuitenkaan sovellu kaikkeen, vaan 2D ja 3D soveltuvat kumpikin paremmin eri tehtäviin [2]. Esimerkiksi 3D soveltuu 2D:tä paremmin hierarkiatasojen esittämiseen [1], sillä esitystapa sallii niiden esittämisen hierarkialle luontaisella tavalla joko päällekäin tai sisäkkäin. 2D puolestaan soveltuu yksityiskohtien, kuten lähdekoodin ja ehtolauseiden, esittämiseen [3]. Parhaimpaan kokonaistulokseen pääsee, kun sekoittaa eri esitystapoja keskenään [2][3].

2.2 Sisällön hallinta

Visualisoinnin haastellisimpia osuuksia on löytää tasapaino esitettävän tiedon ilmaisuvoiman ja tehokkuuden välillä. Kuvaan saa liitettyä sitä enemmän tietoa mitä useampaa parametria sillä voi yhtäaikaisesti mitata. Toisaalta liian monen parametrin yhtäaikainen visualisointi kasvattaa kognitiivista kuormaa, jolloin kuvan kyky välittää tietoa katsojalle laskee. On siis tärkeää rajata olennainen tieto visualisointia varten, jotta kuva pysyy ymmärrettävänä, mutta tehokkaana. [1]

Sisältöä voidaan hallita navigoinnin avulla. Visualisoinnin yhteydessä navigoinnilla tarkoitetaan liikkumista näkymän sisällä. Etenkin 3D-visualisoinnin yhteydessä navigaatio on tärkeimpiä tehtäviä, sillä 3D:ssä lisäulottuvuuden käyttö voi haitata näkyvyyttä objektien jäädessä toistensa taakse. Navigaation avulla voidaan tutkia informaatiota eri perspektiiveistä eri yksityiskohtien tasolta. [2]

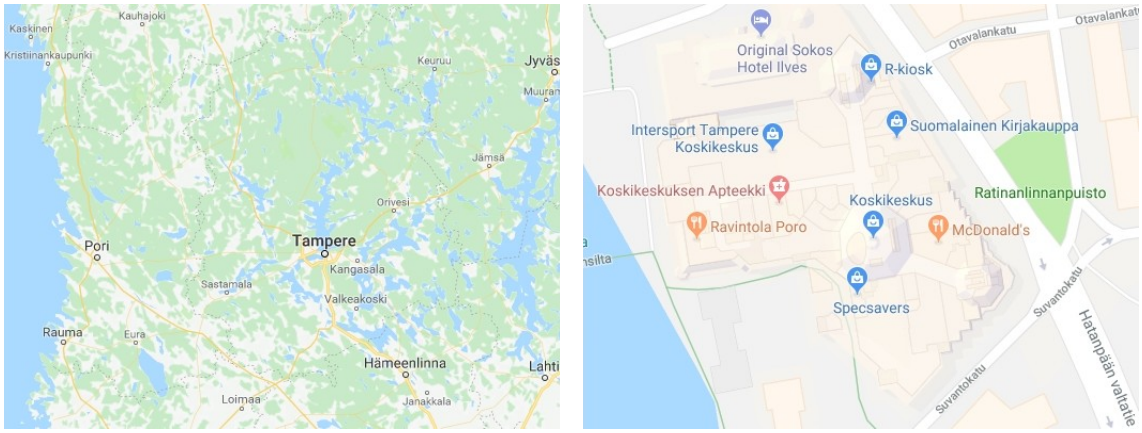
Käyttäjälle on hyvä tarjota ensin kokonaiskuva ja mahdollisuus tutkia yksityiskohtia tarkemmin. Shneiderman esittää informaation haun mantran [8], joka käännettynä kuuluu: ”yleiskuva ensin, zoomaa ja suodata, yksityiskohdat vaadittaessa”. Etenkin

suuria kokonaisuuksia, kuten järjestelmän arkkitehtuuria, visualisoidessa kokonaiskuva auttaa käyttäjää hahmottamaan olennaiset komponentit, jotka vaativat tarkempaa tutkimista [1]. Zoomaamalla ja suodattamalla epäkiinnostavat kohteet näkymästä, käyttäjä voi rajata sekä tutkia itseään kiinnostavia kohteita ja rajatusta näkymästä on myös helpompi tarkastella yksityiskohtia [8].

Zoomaus on yksi oleellisimmista navigaation tehtävistä ja se voidaan toteuttaa eri tavoin. Perinteinen visuaalinen [1][3] zoomi vain suurentaa elementtejä, jolloin zoomaustason kasvaessa yksityiskohdat ovat paremmin esillä, mutta näkymään mahtuvan tiedon määrä vähenee. Tästä edistyneempi muoto on kalansilmäzoomi [1][8], jolla suurennetaan osaa näkymästä ja vastaavasti muita elementtejä voidaan ympäriltä pienentää. Kalansilmäzoomin etu perinteiseen zoomiin on kontekstin parempi säilytys [1], sillä zoomattavan elementin ympäristö pysyy näkyvissä.

Toinen edistynyt zoomausvaihtoehto on semanttinen zoomi [1][2][3], jolla hallitaan esitettävän tiedon määrää ja muotoa. Sen avulla voidaan saman näkymän sisällä esittää paljon informaatiota, mutta se mitä kerralla näytetään riippuu zoomaustasosta. Zoomatessa lähemmäksi karkealla abstraktiotasolla oleva informaatio korvataan tarkemmalla. Tällöin kuvaan saadaan paljon ilmaisuvoimaa, mutta se säilyy myös tehokkaana, koska kerralla näytetään vain zoomauksen tasoon nähden oleellinen tieto. Semanttista zoomausta käytetään esimerkiksi karttasovelluksissa. Kuvassa 5 havainnoidaan Google Mapsin [9] zoomauksen toimintaa: yksityiskohdat, kuten yhtiöiden ja teiden nimet, paljastuvat vasta syvemmälle zoomatessa.

Esteettömän kuvakulman löytäminen, etenkin 3D-näkymässä, voi vaatia kuvakulman kääntämistä ja siirtämistä. Liikkeen myötä ongelmaksi tulee helposti käyttäjän eksyminen. Objektit eivät enää olekkaan siellä missä ne olivat hetki sitten. Yleisenä ratkaisuna eksymisen välttämiseen ehdotetaan navigaation rajaamista [1][2][3]. Rajoittaminen voidaan tehdä liikesuuntiin, esimerkiksi sallimalla vain vaaka- ja



Kuva 5: Semanttista zoomausta käytetään esimerkiksi erilaisissa karttasovelluksissa näyttämään zoomaustasoon nähden oleellinen sisältö.

pystysuuntaiset liikkeet [2], tai estämällä liikkuminen tiettyjen pisteiden yli, kuten ns. maan alle tai rakenteiden lävitse [1]. Rajoitusten ohella voidaan myös hyödyntää automaattisia kamerapolkuja [1][2], jotka pitävät käyttäjän hallitulla reitillä. Koska navigaatio voi kaikesta huolimatta sekoittaa käyttäjän, on hyvä myös pystyä aloittamaan navigaatio alusta [1]. Oli liike sitten zoomausta tai kääntämistä, manuaalista tai automaattista, kun siirtymä tapahtuu pehmeästi käyttäjän on helpompi hahmottaa kontekstin muutosta [8].

Liikkeen rajoituksen ja kamerapolkujen ohella eksymistä voi ehkäistä helpottamalla käyttäjää tunnistamaan ympäristön rakenteita ja auttamalla mielikuvan muodostamista. Tätä voi tehdä esimerkiksi käyttämällä objekteissa eri värejä, tekstuureja ja muotoja. Lisäksi visualisoinnissa voidaan käyttää oikeaan maailmaan pohjautuvaa metaforaa, jolloin järjestelmä kuvataan käyttäjälle tuttuna konseptina, kuten kaupunkina tai aurinkokuntana. [1]

Navigoinnin ohella visualisoinnin yhteydessä on tärkeää pystyä myös interaktion avulla tutkimaan näkymää ja paljastamaan esimerkiksi objekteihin liitettyä lisätietoa. Elementtien valinta suoraan graafisesta kuvasta on nopeaa [8] ja etenkin 3D-näkymissä objekteihin tarraaminen ja niiden siirtäminen raahaamalla on intuitiivista [2]. 3D-järjestelmistä löytyykin esimerkiksi erillisiä kahvoja ja virtuaalikäsiä objektien suoramanipulointiin [2]. Perinteisiin widgetpohjaisiin käyttöliittymiin tottuneen käyttäjän voi kuitenkin olla hankala hahmottaa, että 3D-näkymän objekteille voi suorittaa toimintoja [2].

Valintojen, siirtymien ja muiden ympäristön muutoksien indikointiin voidaan käyttää animaatioita. Sulavat animaatiot, lyhyetkin sellaiset, kiinnittävät katsojan huomion paremmin ja aiheuttavat vähemmän kognitiivista kuormaa kuin äkilliset muutokset sekä tekevät näkymän katselusta miellyttävämpää. [1][2]

3 TEKNOLOGIA

Prototyypin toteutukseen valittiin Qt ja sen 3D-moduuli Qt 3D, sillä prototyyppi halutaan integroida osaksi Kactus2:ta [10], joka on myös toteutettu Qt:lla. Tässä työssä käytetyn Qt:n versio on 5.9.2.

3.1 Qt 3D arkkitehtuuri

Qt 3D on OpenGL:n [11] päälle rakennettu piirto- ja simulaatiokehys. Qt 3D:lla voi hallita kaikkia OpenGL:n ohjelmoitavia piirtoliukuhihnan vaiheita koskematta matalan tason OpenGL ja GLSL toteutukseen [12], vaikka tämäkin on mahdollista. Kuten Qt itsessään, Qt 3D tarjoaa sekä C++- että QML-rajapinnat. Työssä pyrittiin käyttämään pääsääntöisesti C++:aa, mutta myös QML:ää käytettiin.

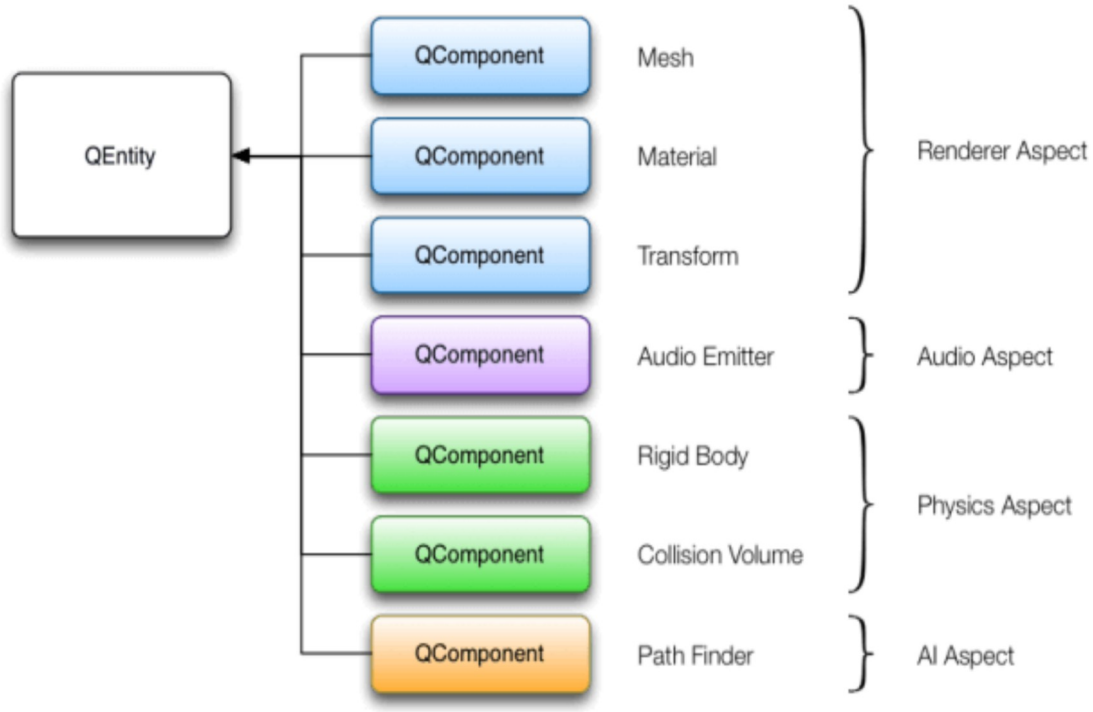
Qt 3D:n toiminta perustuu maisema- (scenegraph) ja kehysgraafiin (framegraph). Maisemagraafi kertoo mitä ohjelman tulisi piirtää, kun taas kehysgraafi kertoo miten se piirretään. [12]

3.2 Maisemagraafi ja aspektit

Maisemagraafi on entiteeteistä koostuva puumainen rakenne, jossa 3D-näkymän olioiden ulkonäkö ja käytös määritetään. Qt 3D:ssä maisemagraafi on toteutettu Entity Component Systemin (ECS) keinoin, jonka rakennetta yhden entiteetin osalta havainnollistetaan kuvassa 6. ECS:ssä 3D-objektit itsessään ovat entiteettejä ja niiden ominaisuudet kuten muoto, sijainti ja käytös ovat komponentteja, jotka periytetään *QComponent*-luokasta. Entiteetit periytetään *QEntity*-luokasta. Komponentit lisätään entiteetteihin heikon koosteen avulla. Koosteen ansioista vältetään monimutkaiset periytymisrakenteet ja mahdollistetaan ajoaikainen komponenttien lisäys ja poisto. [12]

Puun rakenne muodostetaan luomalla entiteettien välillä isä-lapsi-suhteita. Esimerkiksi alla on esitetty pieni maisemagraafi, joka koostuu kahdesta entiteetistä: juuresta ja sen yhden komponentin omaavasta lapsesta. Puussa voi olla useampiakin kerroksia, kunhan ne kaikki muodostavat suorasti tai epäsuorasti lapsisuhteen juureen.

```
Qt3DCore::QEntity* root = new Qt3DCore::QEntity();
Qt3DCore::QEntity* ball = new Qt3DCore::QEntity( root );
Qt3DExtras::QSphereMesh *mesh = new Qt3DExtras::QSphereMesh;
ball->addComponent( mesh );
```



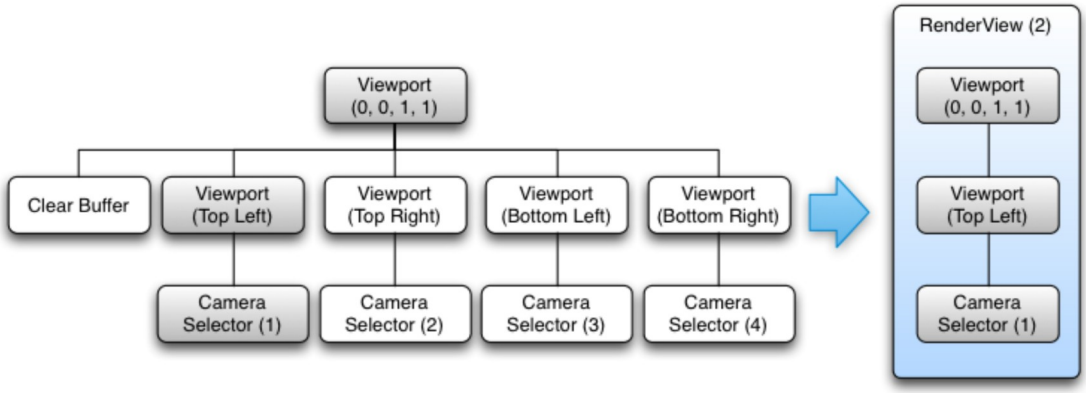
Kuva 6: Qt 3D:n entiteetit toteutetaan ECS:llä. Entiteetti koostuu komponenteista, joiden laskennasta huolehtii niistä vastaava aspekti. [12]

Komponenttien sisältämän datan käsittelystä huolehtivat aspektit, jotka päivittävät entiteetin tilaa sen komponenttien sisältämän datan mukaisesti. Aspektit jaetaan omiin itsenäisiin, esimerkiksi renderöinnistä tai käyttäjän syötteestä vastaaviin kokonaisuuksiinsa. Ohjelman suorituksen aikana aspektit kulkevat maisemagraafia pitkin etsien entiteettejä, joilla on kyseiselle aspektille kuuluvia komponentteja ja suorittavat näihin kohdistuvat laskennat irrallaan muista aspekteista. Tästä johtuen Qt 3D voi skaalauttaa aspektien suorituksen eri ytimille. Qt 3D tarjoaa valmiiksi renderöinti- ja syötteenkäsittelyaspektit, mutta käyttäjä voi toteuttaa siihen myös omia aspekteja. [12]

Sekä entiteettien että komponenttien kantaluokkana on *QObject* [12]. *QObject*-mahdollistaa signaalien ja slottien käytön olioissa. Niiden avulla voidaan ilmoittaa tapahtumista olioiden välillä [13]. Lähdeolio lähettää tapahtumasta signaalin, joka yhdistetään halutun olion slotiin, metodiin, jossa signaalin mukana tulleet parametrit voidaan käsitellä olion haluamalla tavalla.

3.3 Kehysgraafi

Kehysgraafin solmut kertovat renderöijälle yksittäiset askeleet, jotka sen tulee suorittaa yksittäistä kuvaa varten eli piirtotekniikan. Graafilla voidaan määrittää mitä, minne ja



Kuva 7: Esimerkki kehysgraafista, joka piirtää näkymän neljästä eri kamerasta kuvanruudun jokaiseen kulmaan [14]

milloin renderöijän tulee piirtää. Esimerkiksi sillä voidaan määrittää piirretäänkö kuva suoraan näytölle vai välipuskuriin tai missä järjestyksessä entiteetit piirretään. Qt 3D mahdollistaa ajonaikaisen kehysgraafin vaihdon, joten kehysgraafia voidaan muuttaa määrittämällä useita eri graafeja ja vaihtamalla ne tarpeen mukaan. [14]

Renderöijä kiipeää graafia syvyys ensin -järjestyksessä. Kun renderöijä on saavuttanut lehtisolmun, kertoo polku juuresta lehtisolmuun yhden renderöintinäkymän askeleet. Renderöintinäkymä koostaa renderöintikomennot, joiden pohjalta alla toimiva OpenGL piirtää kuvan. Puun haarojen järjestyksellä on väliä ja ensimmäiseksi määritetty haara suoritetaan ensin. Sen sijaan yksittäisen haaran aikaansaama tulos voidaan saavuttaa useammalla eri solmujärjestyksellä. [14]

Kuvassa 7 on esimerkki kehysgraafista, jolla piirretään näkymä neljästä eri kamerasta kuvan jokaiseen kulmaan. Ensimmäisessä renderöintinäkymässä koko kuvan puskuri tyhjennetään. Seuraava renderöintinäkymä, kuvassa harmaalla korostettu, kertoo, että näkymän vasen yläkulma piirretään ensimmäisen kameravalitsimen määrittämän kameran kuvakulmasta. Tämän jälkeen piirretään oikea yläkulma seuraavan kameravalitsimen mukaisesti ja sama tehdään lopuille kulmille. [14]

3.4 Qt3DWindow ja koordinaatisto

Qt3DWindow määrittää valmiiksi konfiguroidun ikkunan, jota voi käyttää 3D-maiseman piirtämiseen. Se käsittää sekä ikkunan fyysisen koon määrittelyn että ikkunan sisällön. Sisältö muodostuu ohjelman käytössä olevista maisema- ja kehysgraafeista. Ikkunan mukana tulee oletuskehysgraafi, *QForwardRenderer* [15], jolla saadaan piirrettyä kameran näkökentässä olevat entiteetit videopuskuriin yksi kerrallaan sävyttämällä jokainen erikseen. Ikkuna pitää sisällään myös *QRenderSettings*-olion [16], jolla

voidaan hallita muita piirron asetuksia, kuten kuvan uudelleenpiirtoa ja poimintametodeja. Tarvittaessa käyttäjä voi ikkunan kautta rekisteröidä toteuttamansa aspektit ohjelman käyttöön. Lisäksi mukana tulee oletuskamera, *QCamera*, jonka kuvakulmasta näkymä piirtyy ikkunaan.

3D-näkymän koordinaatisto muodostuu x-, y- ja z-akseleista. Positiivinen y-akseli osoittaa ylös, x-akseli eteen ja z-akseli oikealle. Vastaavat negatiiviset akselit ovat alas, taakse ja vasemmalle.

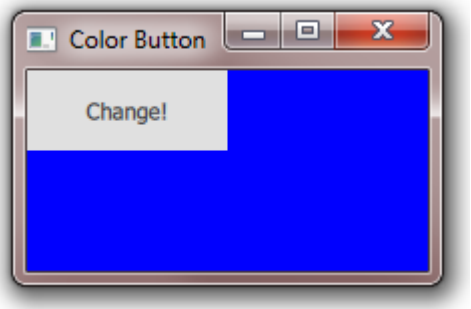
3.5 QML

QML on deklaratiivinen ohjelmointikieli käyttöliittymien toteutukseen. Sen avulla käyttöliittymän rakenne kuvataan visuaalisten elementtien avulla ja elementit voidaan sitoa toisiinsa dynaamiseen tapaan. Kieli on helposti luettavaa ja muistuttaa syntaksiltaan JSON:ia. [17]

```
// main.cpp
#include <QGuiApplication>
#include <QQmlApplicationEngine>
int main(int argc, char *argv[])
{
   QGuiApplication app(argc, argv);
   QQmlApplicationEngine engine;
   engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
   return app.exec();
}

// main.qml
import QtQuick 2.6
import QtQuick.Window 2.2
import QtQuick.Controls 2.1
Window {
   visible: true
   width: 200
   height: 100
   title: qsTr("Color Button")
   Rectangle {
     id: rectangle
     width: parent.width
     height: parent.height
     color: "#0000ff"
     Button {
        text: "Change!"
        onClicked: {
           if ( rectangle.color == "#ff0000" ){
               rectangle.color = "#0000ff"
           } else {
               rectangle.color = "#ff0000"
           }
        }
     }
  }
}
```

Ohjelma 1: Yksinkertaisen QML-sovelluksen toteutus.



Kuva 8: Ohjelman 1 toteuttama käyttöliittymä.

Ohjelmassa 1 on esimerkki yksinkertaisesta QML:llä toteutetusta käyttöliittymästä, jonka nappia painamalla ikkunan täyttävän suorakulmion väri vaihtuu sinisen ja punaisen välillä. Sovellus käynnistetään C++-ohjelman avulla. Kuvassa 8 esitetään ohjelman luoma ikkuna.

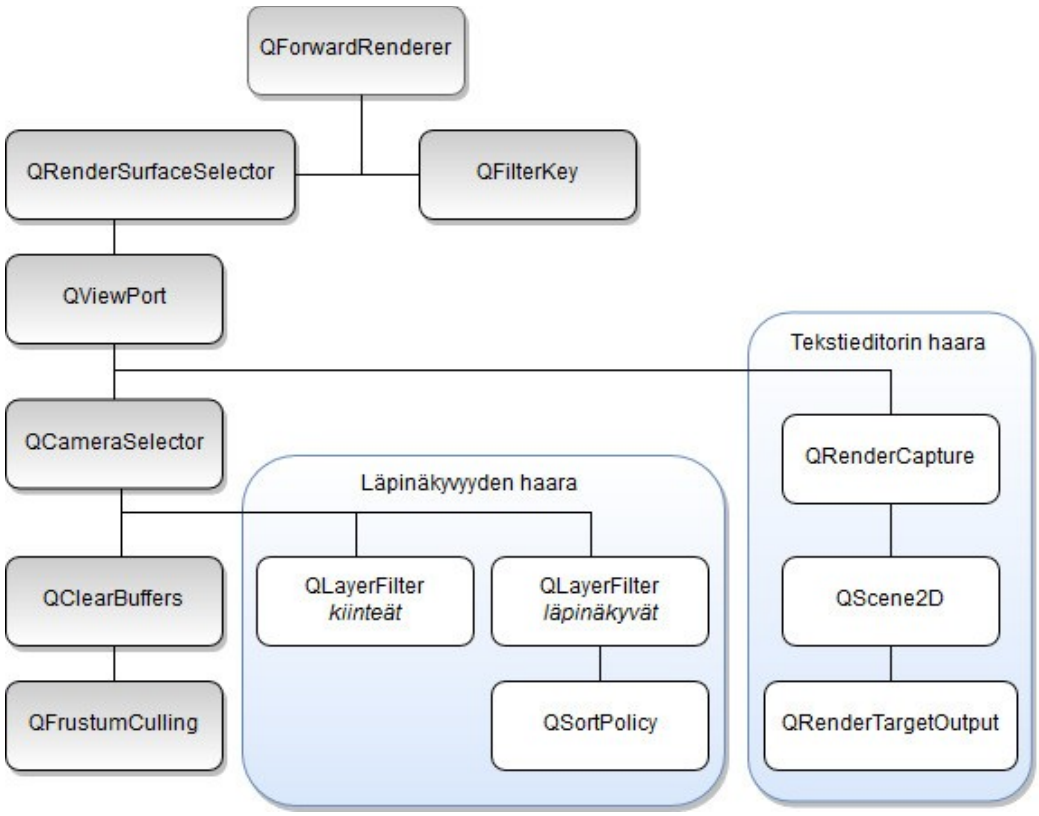
QML:n yhteyteen voidaan lisätä logiikkaa JavaScriptillä, kuten esimerkiksi ohjelman riveillä 27-33 käsiteltävän clicked-signaalin tapauksessa. Sen ohella QML-ohjelmaan voidaan liittää C++-kirjastoja [17].

4 3D-NÄKYMÄN LUONTI

3D-näkymässä kuvataan visualisoitava ohjelma. Se koostuu 3D-valikosta, jolla mallinnetaan ohjelman komponenttien hierarkiarakennetta laatikoina, ja sen ohessa esitetyistä kaksiulotteisista sisänäkymistä, joilla näytetään komponenttien tekstidokumentit.

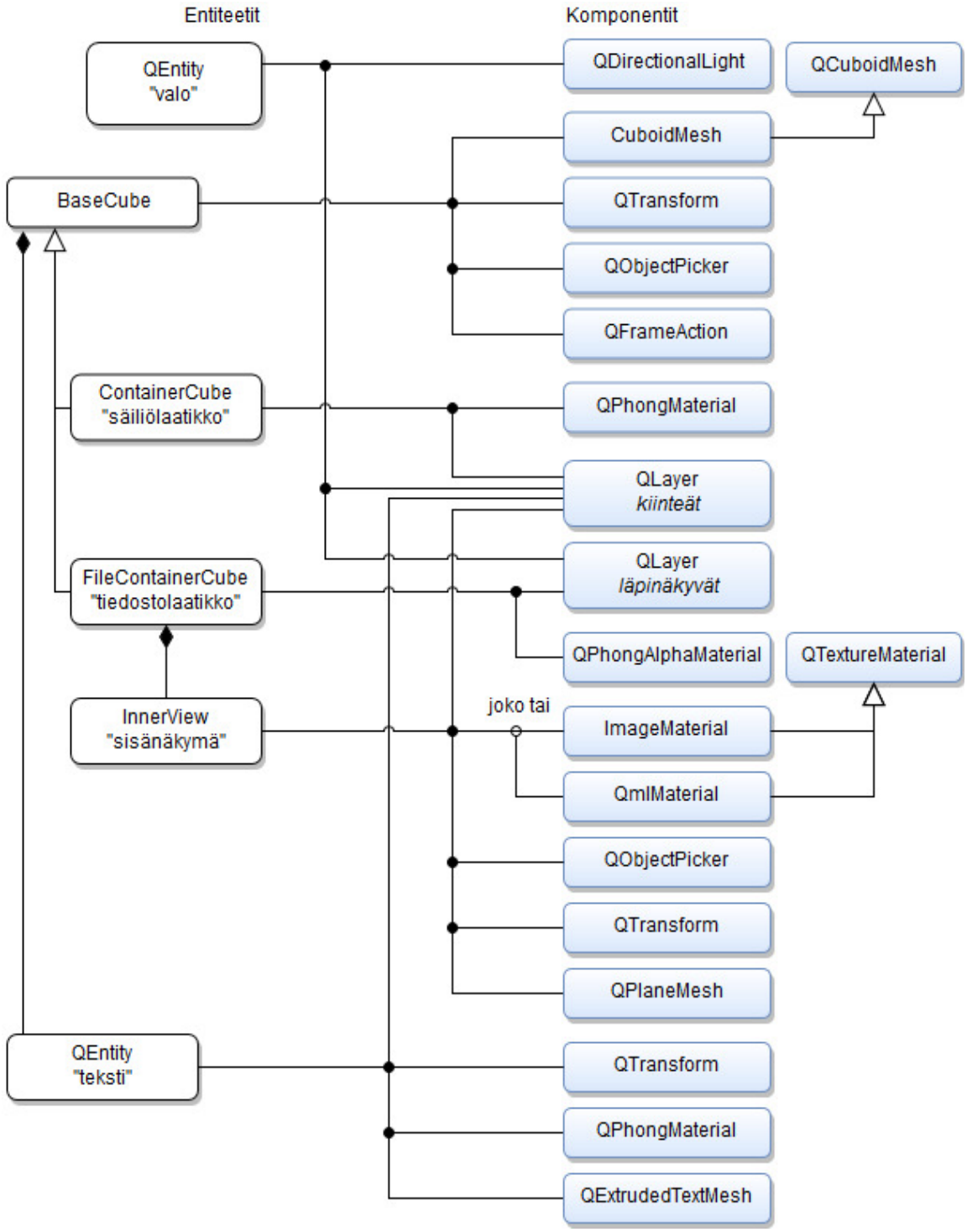
4.1 Yleinen kuvaus arkkitehtuurista

3D-näkymä rakennettiin *Qt3DWindowiin*, jonka mukana tulevaa kehysgraafia muokattiin työtä varten. Prototyyppissä käytetty kehysgraafi on esitetty kuvassa 9. Kehysgraafin harmaa haara tulee *QForwardRendererin* mukana. Loput osat graafista ovat prototyyppiä varten tehtyjä muokkauksia ja ne on esitelty alaluvuissa 4.2.1 ja 5.3.



Kuva 9: Työssä käytetty kehysgraafi on rakenttettu Qt3DWindowin oletuskehysgraafin päälle.

Maisemagraafi koostuu näkyvistä entiteeteistä ja kontrollientiteeteistä. Näkyviä entiteettejä ovat 3D-valikon laatikot, niiden nimikyltteinä käytetyt tekstit, sisänäkymät ja valo. Näkyvien entiteettien komponenttirakenne on esitetty kuvassa 10. Kontrollientiteetit eivät näy ja niiden tarkoitus on huolehtia sisänäkymän sisällöstä sekä kameran liikkeistä. Kontrollientiteettejä käsitellään tarkemmin luvussa 5.



Kuva 10: Näkyvät entiteetit muodostuvat laatikoista, niiden sisänäkymistä ja teksteistä sekä valosta.

4.2 3D-valikko

3D-valikko koostuu tiedostolaatikoista, jotka pitävät sisällään sisänäkymän. Sisänäkymää käytetään piirtämään tekstieditori ja editorin sisältönä käytettävä tiedosto määräytyy sen mukaan, mitä tiedostoa sisänäkymää ympäröivä tiedostolaatikko vastaa. Lisäksi valikossa on säiliölaatikoita, jotka esittävät kuvattavan arkkitehtuurin moduuleita. Ne pitävät sisällään tiedostolaatikoita sekä muita säiliölaatikoita. Kuten alkuperäisessä prototyypissä, siirtymä kolmiulotteisesta näkymästä kaksiulotteiseen haluttiin toteuttaa läpinäkyvyyden avulla.

4.2.1 Läpinäkyvyys

Läpinäkyvyys on materiaalin ominaisuus ja sitä ilmaistaan alfa-arvolla. Alfan arvo on välillä [0,1]. Kun objekti on täysin läpinäkyvä, toisin sanoen näkymätön, alfan arvo on 0 ja täysin kiinteän värisen, läpinäkymättömän, alfa on 1. [18, s. 199]

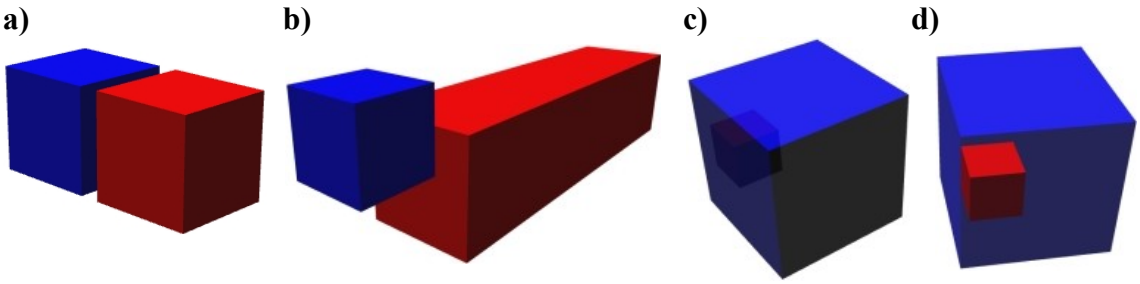
Loppullinen näytön pisteen väri määräytyy *sekoittamalla* (blending) pisteen kohdalla olevien objektien väri niiden alfojen avulla. Objektit piirretään yksitellen puskuriin ja seuraavan objektin värityksessä otetaan huomioon puskurissa jo ennestään oleva väri. Uusi väri lasketaan kaavalla

$$C_{final} = \alpha_{src} C_{src} + (1 - \alpha_{src}) C_{dst} , \quad (1)$$

jossa C_{final} on uusi sekoitettu väri, α_{src} on pikseliehdokkaan alfa, C_{src} pikseliehdokkaan väri, C_{dst} puskurissa oleva väri. [18, s. 200]

Läpinäkyvyyden piirto sekoituksen avulla on järjestyksestä riippuvainen. Läpinäkyvät objektit on piirrettävä taaimmainen ensiksi, jotta väri sekottuisi oikein. Lisäksi läpinäkymättömät objektit on piirrettävä ennen läpinäkyviä. Kiinteän värisiä objekteja piirtäessä ei tarvitse välittää järjestyksessä, mutta niitä piirtäessä on käytettävä syvyystestausta, jotta niiden takana olevat läpinäkyvät objektit eivät piirry niiden päälle. Läpinäkyvillä syvyystestaus otetaan pois käytöstä, jotta kaikki pikselin lopulliseen väriin vaikuttavat objektit otetaan huomioon loppuväriä laskettaessa. [18, s. 200–201]

Järjestäminen etäisyyden mukaan ei aina ole yksiselitteistä. Entiteettipohjaisissa järjestelmissä järjestäminen tehdään koko entiteetin sijainnin mukaan. Siinä ei oteta huomioon minkä muotoinen tai kokoinen entiteetti on, joten järjestys menee helposti väärin. Tätä on havainnollistettu kuvassa 11, jossa laatikot piirretään kauimmainen ensin sekoituksen avulla, kun laatikon sijaintina on sen keskipiste.



Kuva 11: Järjestäminen etäisyyden mukaan ei ole aina yksiselitteistä. a) Sininen laatikko on kauempana. b) Pidemmän muodon vuoksi punainen laatikko on kauempana. c) Sisäkkäisistä laatikoista punainen on kauempana. d) Kohdan c laatikot toisesta kuvakulmasta, jolloin sininen on kauempana.

Monimutkaisten näkymien järjestämiseen löytyy muutama ratkaisu [18, s. 205]. Joko objektien monikulmioverkot järjestetään kolmioiden mukaan tai ongelmalliset kohdat ratkaistaan rikkomalla objekti useampaan järjestettävissä olevaan osaan. ECS:ssä tämä on hankalaa, sillä entiteetit käsitellään kokonaisuuksina. Jos monikulmioverkkoa haluttaisiin jakaa, täytyisi entiteettikin jakaa useaan entiteettiin.

Eräs vaihtoehto läpinäkyvyyden renderöintiin on järjestämisestä riippumaton *syvyyskuorinta* (depth-peeling). Siinä koko näkymä käsitellään kokonaisuutena kapea kerros kerrallaan syvyyssuunnassa ja näistä kerroksista renderöidään lopullinen näkymä objektikohtaisen renderöinnin sijaan. [18, s. 205]

Alkuperäisessä suunnitelmassa oli tarkoituksena toteuttaa myös säiliölaatikot läpinäkyvyyden avulla siten, että sisemmät laatikot paljastuvat samaan tapaan kuin sisänäkymä tiedostolaatikoiden sisältä. Koska säiliöt olisi hankala järjestää eri muotojen ja sisäkkäisten rakenteiden takia, jätettiin läpinäkyvyys vain tiedostolaatikoille. Tiedostolaatikoiden kohdalla järjestäminen on aina yksiselkoista niiden asettelun takia. Laatikoiden asettelua käsitellään tarkemmin seuraavassa luvussa.

Qt 3D:stä löytyy valmiiksi yksi sekoitusta tukeva materiaali, *QPhongAlphaMaterial*. Tällä komponentilla entiteettille tehdään phong-sävytys. Muut entiteetit käyttävät *QPhongMaterialia*, joka tekee samanlaisen sävytyksen, mutta se ei sisällä läpinäkyvyystukea. Sekoitusfunktiot määritetään osana materiaalia. Ohjelmassa 2 lisätään tiedostolaatikolle komponentiksi *QPhongAlphaMaterial*, jonka sekoitusfunktiot saavat aikaan kaavan 1 tuloksen.

```
1  void FileContainerCube::setMaterial()
2  {
3    Qt3DExtras::QPhongAlphaMaterial* material = new Qt3DExtras::QPhongAlphaMaterial( this );
4    material->setAlpha( 0.5f );
5    material->setDiffuse( getCubeColor() );
6    material->setSpecular( getCubeColor() );
7    material->setShininess( 1.0 );
8    material->setAmbient( QColor( 50, 50, 50) );
9
10   material->setBlendFunctionArg( Qt3DRender::QBlendEquation::BlendFunction::Add );
11   material->setSourceAlphaArg( Qt3DRender::QBlendEquationArguments::SourceAlpha );
12   material->setDestinationAlphaArg(Qt3DRender::QBlendEquationArguments::OneMinusSourceAlpha);
13   addComponent( material );
14 }
```

Ohjelma 2: QPhongAlphaMaterialin määrittely värin ja sekoitusfunktioiden osalta ja lisäys entiteetin komponentiksi.

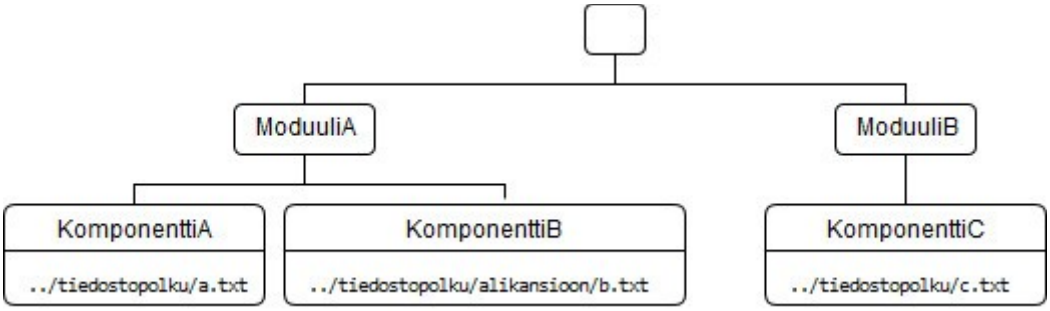
Piirron järjestystä hallitaan kehysgraafin kerrossuodattimilla, *QLayerFiltereillä*. Valikon näkyvät entiteetit jaetaan kahteen ryhmään niiden materiaalin perusteella: kiinteän värisille entiteeteille annetaan komponentiksi *QLayer*, joka kuuluu kuvassa 9 näkyvälle vasemmanpuoleiselle kerrossuodattimelle. Vastaavasti läpinäkyville annetaan oikeanpuoleisen kerrossuodattimelle kuuluva kerros. Valoentiteetille annetaan molempien suodatinten kerrokset, jotta valo vaikuttaisi kaikkiin entiteetteihin. Jälkimmäisellä kerrossuodattimella on lapsena *QSortPolicy*, jolla voidaan määrittää kerrossuodattimen vaikutuksenalaisille entiteeteille järjestämissuunta, tässä tapauksessa kauimmainen ensiksi.

Koska renderöijä käy kehyspuun haarat syvyys ensin-järjestyksessä ja erilliset haarat lisäysjärjestyksessä (kuvassa 9 vasemmalta oikealle), piirretään ensiksi ensimmäistä kerrossuodatinta vastaavat entiteetit. Koska kehysgraafissa ei ole erikseen määritetty *QRenderStateSetiä*, otetaan käyttöön automaattisesti syvyystesti [19]. Näiden jälkeen piirretään toisen kerrossuodattimen entiteetit järjestyskäytännön määrittämällä tavalla. *QPhongAlphaMaterialin* yhteydessä käytetään automaattisesti *QNoDepthMaskia* [20], joka estää syvyyspuskurin päivittämisen. Koska järjestämistä ei alkuperäisen prototyypin yhteydessä oltu suoritettu, piirtyivät entiteetit niiden lisäysjärjestyksessä ja lopputuloksena oli kuvassa 2 näkyvä ongelma, jossa entiteetit piirtyivät väärässä järjestyksessä.

4.2.2 Hierarkiarakenteen järjestäminen

Moduulien hierarkiarakenne luodaan erillisestä asetustiedostosta, jossa on listaus kuvattavan ohjelman tiedostoista. Esimerkki kolmen komponentin asetustiedoston sisällöstä on alla:

```
ModuuliA/KomponenttiA;../tiedostopolku/a.txt
ModuuliA/KomponenttiB;../tiedostopolku/alikansioon/b.txt
ModuuliB/KomponenttiC;../tiedostopolku/c.txt
```



Kuva 12: Yksinkertaisen asetustiedon pohjalta muodostettu komponenttien hierarkiarakenne.

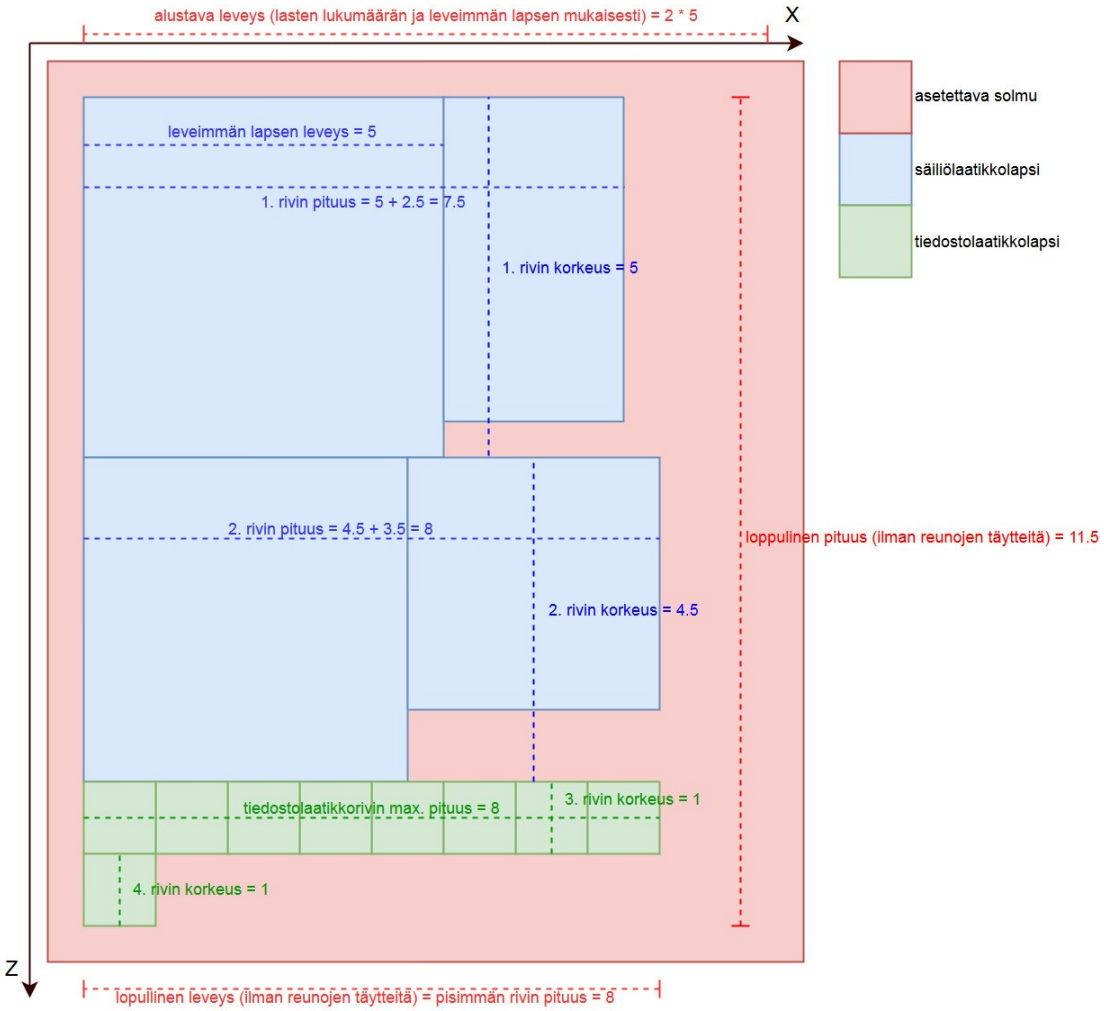
Komponentista on tallennettu sen sijainti hierarkiassa ja sitä vastaavan dokumentin sijainti kovalevyllä erotettuna puolipilkulla. Koska fyysinen sijainti ja sijainti hierarkiassa ovat toisistaan riippumattomia, ei tiedostojen tarvitse olla tallennettu fyysisesti hierarkiarakenteen mukaisesti.

Hierarkiatiedoston pohjalta muodostetaan puu, joiden solmut kuvaavat hierarkiarakennetta. Kuvassa 12 on edellisen asetustiedoston pohjalta muodostettu rakenne. Lehtisolmuina ovat yksittäiset komponentit, joille on määritetty tiedostot ja muut solmut vastaavat moduuleita. Kun hierarkiapuu on muodostettu, luodaan solmuille asetelma, joka näkyy 3D-näkymässä.

Asetelmassa pyritään siihen, että laatikot on aseteltu tiiviisti, mutta siten etteivät ne olisi toistensa tiellä. Asetelma tehdään ruudukon avulla, jonka perusyksikkö vastaa yhden tiedostolaatikon kokoa. Säiliölaatikoiden koko määrittyy niiden sisällä olevien tiedostolaatikoiden ja muiden säiliöiden mukaisesti.

Asettelu tehdään syvyys ensin -järjestyksessä. Solmun lapsista etsitään ensiksi levein lapsi, jonka pohjalta solmun alustava leveys määritellään. Mikäli lapsen pituus (mitta z-akselilla) on suurempi kuin leveys (mitta x-akselilla), käännetään lapselle muodostettu asetelma päikseen x- ja z-akselin suhteen, jolloin leveyttä tarkastellaan uuden leveyden, eli vanhan pituuden, suhteen.

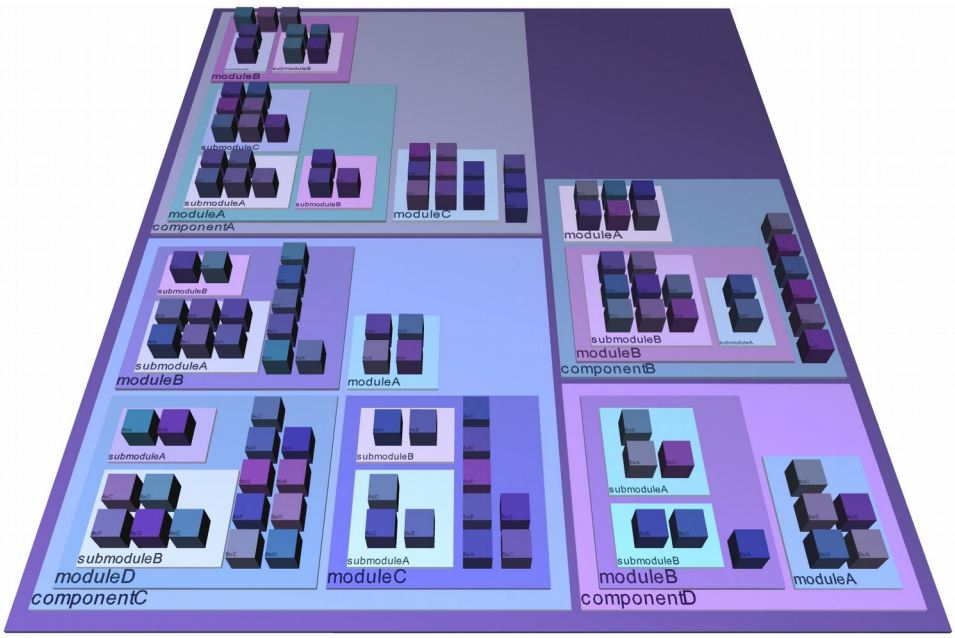
Kun levein lapsi on löydetty, varataan solmun asetelmalle alustava leveys. Alustava leveys on $w_{max}\lceil\sqrt{(c_{count})}\rceil$, jossa w_{max} on suurimman lapsen leveys ja c_{count} lasten lukumäärä. Mikäli lapsissa on säiliölaatikoita, käytetään lasten määränä vain säiliölaatikkolasten lukumäärää. Tällä pyritään siihen, että solmun lopullisen asetelman leveys ja pituus on mahdollisimman lähellä toisiaan. Mikäli säiliölaatikoiden lisäksi varattaisiin tilaa tiedostolaatikoille suurimman lapsen mukaisesti, päädytään helposti pitkulaiseen asetelmaan, joka ei näytä yhtä miellyttävältä.



Kuva 13: Solmun sisäisen rakenteen asettelu. Alustava leveys määräytyy leveimmän säiliölaatikon ja säiliölaatikoiden lukumäärän mukaan. Lopullinen leveys määräytyy leveimmän laatikkorivin mukaan ja pituus kaikkien rivien yhteispituuden mukaan. Järjestäminen aloitetaan vasemmasta yläkulmasta.

Seuraavaksi luodaan solmun sisäinen asetelma, jota on havainnollistettu kuvassa 13. Lapset järjestetään ensin pituusjärjestykseen suurin ensiksi. Tämän jälkeen ne asetetaan riveittäin solmun sisälle alustavan leveyden asettaman rajoitteen mukaisesti. Tässä vaiheessa lapsen koordinaatit määritetään suhteessa solmun koordinaatteihin. Kun kaikki säiliölaatikot on saatu aseteltua, leikataan solmun leveys leveimmän rivin mukaisesti ja tiedostolaatikot aloittavat uuden rivin. Lasten järjestämisen jälkeen lisätään solmulle vielä reunoille täydennykset, jotta sisäkkäiset rakenteet on helpompi erottaa lopullisesta asetelmasta.

Kun jokainen hierarkiapuun solmu on aseteltu, lasketaan x- ja z-koordinaatit, joiden pohjalta 3D-näkymä luodaan. Tässä vaiheessa määritetään myös laatikon pinoutumiskorkeus, joka määrittää laatikon sijainnin y-akselilla. Kun varsinainen näkymä luodaan, lasketaan laatikon keskipiste edellä laskettujen koordinaattien ja



Kuva 14: Esimerkki lopullisesta asetelmasta, johon on lisätty laatikoiden väliset sisennykset, pinoutumiskorkeus ja komponenttien nimet.

laatikon mittojen avulla. Tässä vaiheessa lisätään myös toisiinsa kiinni olevien laatikoiden väliin raot.

Laatikon sijainti 3D-näkymässä määritetään *QTransform*-komponentilla, joka huolehtii entiteetin sijainnista, orientaatiosta ja skaalauksesta. Leveyden, pituuden ja pintoutumiskorkeuden avulla luodaan monikulmioverkko, *CuboidMesh*, joka luo annettujen mittojen mukaisen 3D-laatikon.

Esimerkki valmiiksi asetellusta 3D-näkymästä on esitettynä kuvassa 14. Lopullisen asettelun yhteydessä lisätään vielä hierarkiarakenteessa määritetty solmun nimi laatikon reunalle. Teksteissä on käytetty *QExtrudedTextMeshiä*, jolla luodaan kolmiulotteisia tekstejä. Sen lisäksi, että teksti selventää mistä komponentista laatikon kohdalla on kyse, sen tarkoituksena on myös toimia maamerkkinä: teksti sijaitsee aina samalla reunalla laatikoissa, joten käyttäjän on helpompi hahmottaa mistä suunnasta näkymää katsotaan.

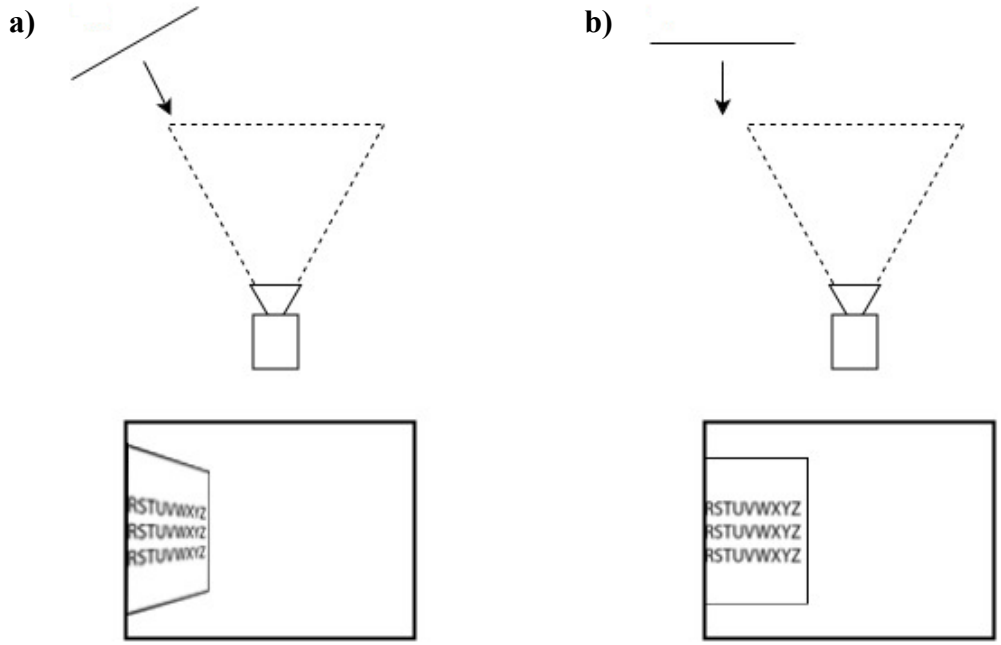
4.2.3 2D-näkymän esittäminen 3D-valikon yhteydessä

Sisänäkymät esitetään mainostauluina. Mainostaulut ovat objekteja, jotka muuttavat asentoaan katsomaan katsojaa kohti tai katsojan kuvaustason suuntaisesti. Perinteisesti mainostauluja käytetään esittämään objekteja, jotka ovat symmetrisiä jonkin akselinsa ympäri tai joiden geometriaa halutaan yksinkertaistaa. [18, s. 257–261][21, s. 216–217]

Riippuen siitä, kuinka objekti on akseliensa mukaisesti symmetrinen, kääntäminen tapahtuu yhden tai useamman akselin ympäri. Jos objekti on sylinterimäisesti symmetrinen, kuten puu, käännetään sitä yleensä yhden akselin, useimmiten ylösvektorinsa ympäri. Jos objekti taas on pallomaisesti symmetrinen, kuten pilvi tai räjähdys, käännetään sitä sekä pysty- että sivusuunnassa. Käyttötapauksesta riippuen, mainostaulun ylösvektori voi olla maailman tai katsojan mukainen. [18, s. 257–261] [21, s. 216–217]

Geometrioiden yksinkertaistamiseen mainostauluja voidaan käyttää siten, että objektin monikulmioverkko esitetään monimutkaisempana kuin mitä se oikeasti on. Esimerkiksi pallo voidaan monikulmioverkon puolesta toteuttaa puolikkaana pallona. Mainostalulutuksen avulla kaareva pinta pidetään aina katsojaan päin, jolloin pallo näyttää kokonaiselta. Geometriaa voidaan yksinkertaistaa myös niin paljon, että monikulmioverkko korvataan tasolla ja objekti saa yksityiskohtainsa tekstuurissa olevan kuvan avulla. Tämä on etenkin hyödyllinen tilanteessa, jossa objekti on aina kaukana katsojasta. [18, s. 257–261]

Alkuperäisessä prototyypissä sisänäkymä kääntyi katsomaan aina kameraa kohti, kuvan 15a mukaisesti. Tässä käyttötapauksessa esitystapa on huono, sillä tekstin koko vaihtelee tason eri osissa, jos kamera ei katso suoraan tasoa kohti. Parempi esitystapa on kääntää taso kameran kuvaustason suuntaisesti kuvan 15b mukaisesti, jolloin tasoon ei tule perspektiivistä johtuvia vääristymiä ja teksti on paremmin luettavissa.



Kuva 15: a) Jos sisänäkymä katsoo aina kohti kameraa, on tekstiä hankala lukea perspektiivistä johtuvan vääristymän vuoksi. b) Sen sijaan kääntäminen kuvaustason mukaisesti on suotavampaa luettavuuden kannalta.

Sisänäkymä käyttää monikulmioverkkonaan tasoa, *QPlaneMeshiä*. Koska sisänäkymän halutaan olevan luettavissa mistä suunnasta tahansa, käännetään sitä sekä pysty- että vaakasuunnassa. Tämä tapahtuu kuuntelemalla kameran *upVectorChanged*- ja *viewVectorChanged*-signaaleja, joita kamera lähettää aina kun sen ylösvektori- tai katsevektori muuttuvat kameran kääntämisen tai liikuttamisen yhteydessä. Kun toinen vektoreista päivittyy, otetaan signaalin mukana tullut vektori talteen ja lasketaan sisänäkymälle uusi rotaatio ohjelman 3 mukaisesti.

Tasolle asetetaan leveys ja korkeus, jotka määritetään siten, että niiden suhde on sama kuin ikkunan, johon koko näkymä piirretään. Tällöin editori saadaan täyttämään koko ruutu siirryttäessä editoritilaan, eikä ympärille jää rakoja, joista 3D-näkymää voisi näkyä. Lisäksi *QPlaneMeshille* määritetään peilataanko sen UV-koordinaatit, joiden mukaan tekstuuri asetetaan monikulmioverkkoon. Peilaus asetetaan todeksi, koska muutoin kuva piirtyy tasolle väärin päin.

```
1  void InnerView::changeRotation()
2  {
3     Qt3DCore::QTransform* transform = getTransformComponent();
4     if( transform == nullptr )
5     {
6        return;
7     }
8
9     QVector3D lookAtDirection = -cameraViewVector_;
10    lookAtDirection.normalize();
11    QQuaternion rotation = QQuaternion::fromDirection( lookAtDirection, cameraUpVector_ );
12    QQuaternion xRotation = QQuaternion::fromAxisAndAngle( 1, 0, 0, 90 ); //nosto pystyasentoon
13    getTransformComponent()->setRotation( rotation * xRotation );
14 }
15
16 void InnerView::onCameraUpVectorChanged( QVector3D cameraUpVector )
17 {
18    cameraUpVector_ = cameraUpVector;
19    changeRotation();
20 }
21
22 void InnerView::onCameraViewVectorChanged( QVector3D cameraViewVector )
23 {
24    cameraViewVector_ = cameraViewVector;
25    changeRotation();
26 }
```

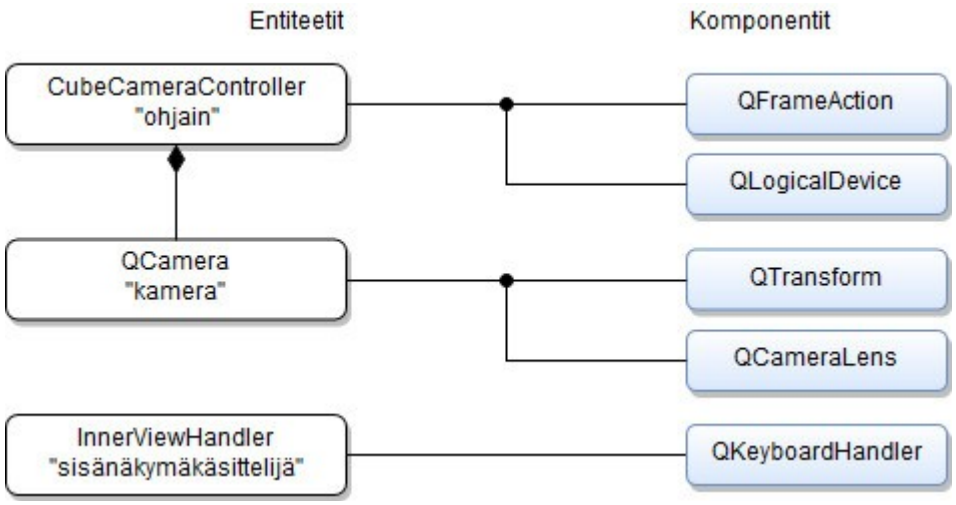
Ohjelma 3: Sisänäkymä kääntyy aina kameran ylös- ja katsevektorien suuntaisesti.

5 NAVIGOINTI JA MUOKKAUS

Navigoinnilla pyritään siihen, että käyttäjä voi katsella näkymää eri kuvakulmista eri yksityiskohtien tasolta. Lisäksi halutaan ehkäistä eksymistä. Tekstidokumenttien muokkaus tehdään sisänäkymien kautta. Navigointia ja muokkausta varten lisättiin kontrollientiteetit, joiden komponenttirakenteet ovat kuvassa 16.

5.1 Liikkuminen 3D-valikossa

Liikkuminen tarkoittaa ohjelman näkökulmasta kameran liikuttamista näkymässä. Kameran liikkeistä vastaa ohjain, joka liikuttaa kameraa päivittämällä sen *QTransform*-komponenttia käyttäjän syötteiden mukaisesti. Kamera on *Qt3DWindowilta* saatu kamera. *QFrameAction* on logiikkakomponentti, joka lähettää *triggered*-signaalin jokaisen kuvanpiirron välissä. Signaalin parametrina saadaan aika sekunteina, delta-aika, joka on kulunut edellisen kuvan piirrosta. Delta-ajan avulla määritetään, kuinka pitkän matkan kamera liikkuu tai kääntyy kuvien välillä: mitä pidempi delta-aika, sitä pidempi liike. Tällä varmistetaan se, että liike etenee tasaisesti vaihtelevilla kuvataajuuksilla. *QLogicalDevice* vastaa hiiri- ja näppäinsyötteiden kuuntelusta ja *QCameraLens* määrittää kameran näkökentän. Sisänäkymäkäsittelijään perehdytään luvussa 5.3.



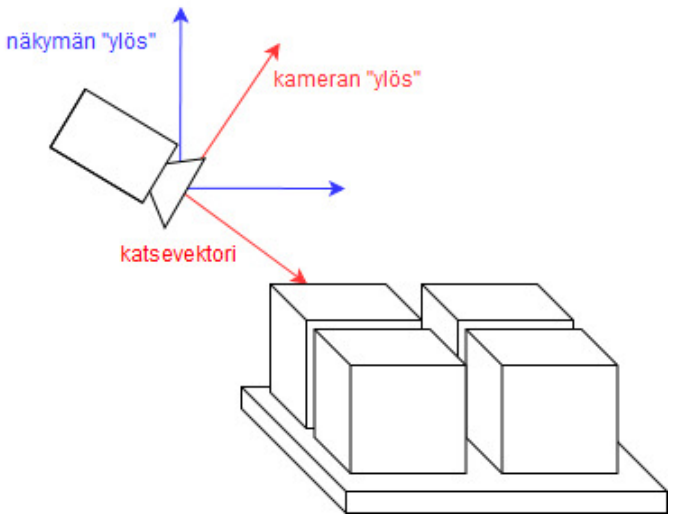
Kuva 16: Kontrollientiteetit koostuvat kameraohjaimesta sekä sisänäkymäkäsittelijästä.

5.1.1 Liikkuminen tiedostolaatikon ulkopuolella

Liikkumista voidaan hallita sekä hiiri- että näppäinsyötteillä. Liikkeet, jotka toteutetaan, ovat siirtymä eteen ja taakse, sivulle, ylös ja alas sekä kameran kääntäminen vaaka- ja pystysuunnassa. Tätä varten *QLogicalDevicelle* määritetään laitteet, joita halutaan kuunnella, eli tässä tapauksessa hiiri ja näppäimistö. Molemmille rekisteröidään toiminnot ja akselit, joilla kameraa liikutetaan. Toimintoja ovat eri näppäinten ja hiiren painikkeiden syötteet, akseleita puolestaan hiiren liike pysty- ja vaakasuoraan, keskirullan liike, sekä näppäimistön nuolinäppäinten syötteet.

Eteen- ja taaksepäin tapahtuva siirtymä tehdään hiiren keskirullalla tai näppäimistön ylä- ja alanuolilla. Hiirellä siirtyessä kamera siirtyy katsevektorin suuntaisesti. Lisäksi kursorin sijainnilla pystytään vaikuttamaan hieman liikkeen suuntaan: Jos kursori on lähellä näytön reunoja, kameraa käännetään hieman hiiren osoittamaan suuntaan. Näppäimistöllä liikuttaessa kursorin sijainnilla ei ole merkitystä, vaan kamera liikkuu aina eteenpäin. Näppäimistön kohdalla on myös intuitiivisempaa, jos kameran y-sijainti pysyy samana liikkeen ajan.

Pysty- ja sivusuuntainen siirtymä tapahtuu hiiren vasemman painikkeen avulla. Kun painaa hiiren vasenta painiketta ja liikuttaa hiirtä, kamera siirtyy itseensä nähden pysty- tai sivusuunnassa hiiren liikkeen mukaisesti. Näppäimistöllä sivusuuntainen liike tapahtuu vastaavalla tavalla vasemmalla ja oikealla nuolinäppäimellä. Pystysuuntaisessa liikkeessä käytetään myös ylös- ja alasnäppäimiä, mutta eteen- ja taaksepäin liikkeestä ne erotetaan painamalla control-näppäintä. Tässäkin tapauksessa tuntuu luotevammalta, että näppäimistöllä ”ylös” tarkoittaa näkymän mukaista ”ylös”-suuntaa. Hiirellä ja näppäimistöllä tapahtuvan liikkeen eroja on havainnollistettu kuvassa 17.



Kuva 17: Ero hiirellä (punainen) ja näppäimistöllä (sininen) tapahtuvan liikkeen välillä. Hiirellä ylös- ja eteenpäin liike riippuvat kameran kallistumasta. Näppäimistöllä ylöspäin tarkoittaa näkymän ”ylös”-suuntaa ja eteenpäin pitää y-sijainnin samana liikkeen aikana.

Kameran siirtymänopeus on sidottu myös kameran y-sijaintiin. Tällä mahdollistetaan se, että tarkasteltessa näkymää korkealta tasolta, voidaan siirtyä nopeasti pitkiä etäisyyksiä, kun taas matalalla tasolla samassa ajassa siirrytään huomattavasti lyhyempiä etäisyyksiä. Näin kamera pysyy käyttäjän hallinnassa eri tasoilla, mutta käyttäjää ei myöskään pitkästytetä hitaalla liikkeellä.

Kameran kiertäminen ja kallistaminen tapahtuu hiirellä oikealla painikkeella ja liikuttamalla hiirtä haluttuun suuntaan. Näppäimistöllä kääntäminen tehdään nuolinäppäimillä haluttuun suuntaan, mutta ne erotetaan siirtymisestä painamalla shift-näppäintä.

Kameran liikkeet on rajattu alimman säiliölaatikon mukaan. Kamera saa liikkua vain hieman laatikon reunojen yli sivusuunnassa, pystysuunnassa kamera saa liikkua vain laatikon yläpuolella, sielläkin rajoitettuun korkeuteen asti. Tällä estetään käyttäjää eksymästä hierarkiarakenteen ympärillä olevaan tyhjään alueeseen.

Kamera on myös mahdollista palauttaa aloitusnäkymään siltä varalta että käyttäjä eksyy ja haluaa aloittaa navigoinnin alusta. Palautus voidaan aloittaa painamalla hiiren keskirullaa tai escape-näppäintä. Tällöin kameran siirtyy autopilottitilaan palautuen suorinta reittiä alkupisteeseen ja kääntää katseen kohti hierarkiarakenteen keskipistettä.

Navigointi aloitetaan siten, että 3D-valikko on keskitetty näkymän keskelle ja kamera kuvaa sitä korkealta. Näin tarjotaan heti navigoinnin aloittamisen yhteydessä käyttäjälle kokonaiskuva valikosta. Palautuspiste navigoinnin uudelleenaloituksessa on sama kuin aloitusnäkymä, joten käyttäjä palaa itselleen tuttuun näkymään.

5.1.2 Liikkuminen tiedostolaatikon sisäpuolella

Kun kameran sijaintia muutetaan, se lähettää *positionChanged*-signaalin. Jokainen laatikko kuuntelee kyseistä signaalia, ja signaalin saatuaan, tarkistetaan sen parametrina saadun kameran sijainnista, onko kamera laatikon sisällä. Jos on, laatikko lähettää signaalin, jonka parametrina se antaa osoittimen omaan sisänäkymäänsä. Ohjain saa signaalista tiedon olevansa laatikon sisällä, jolloin se tallentaa nykyisen sijaintinsa, eli sisääntulopisteensä, sekä signaalin mukana saadun sisänäkymän, eli katselukohteensa.

Tiedostolaatikon sisällä kameran liike on automaattista. Ensiksi kameran katsevektori liu'utetaan kohti katselukohteen keskipistettä. Kun katse on suoraan keskipisteessä, siirretään kamera kohti sisänäkymää niin kauan, kunnes sisänäkymä täyttää koko kuvaruudun. Tällöin käyttäjä on siirtynyt editoritilaan.

Editoritilasta poistutaan painamalla joko escape-näppäintä tai hiiren keskirullaa. Poistuessa kamera säilyttää katseen kohti katselukohdettaan ja peruuttaa kohti sisääntulopistettä. Kun kamera on tiedostolaatikon ulkopuolella, palautetaan ohjaus käyttäjälle.

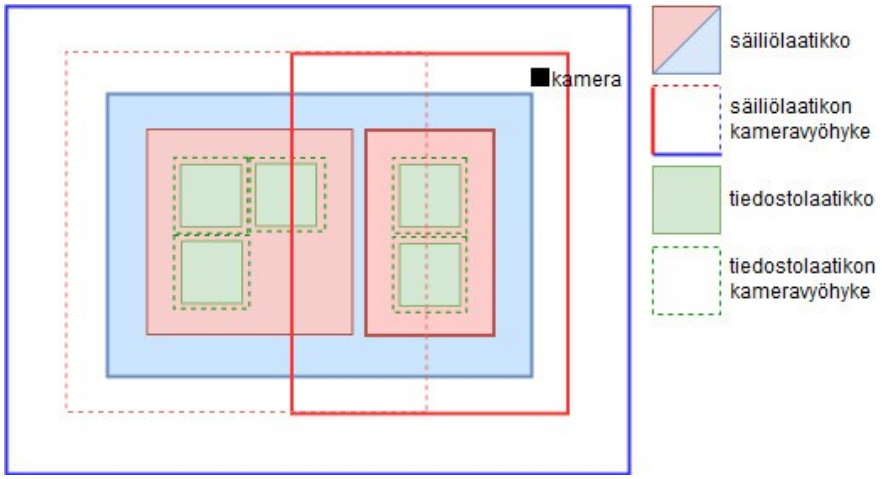
5.2 Sisällön rajaaminen

Koska laatikoita voi näkymässä olla jopa satoja, voi näytettävää sisältöä olla paljon. Yksityiskohtien määrä halutaan pitää jatkuvasti mahdollisimman vähäisenä, joten oletusarvoisesti säiliölaatikoiden sisältö on piilotettu. Näytettävää sisältöä hallitaan pääasiassa semanttisella zoomauksella, mutta tarvittaessa käyttäjä voi paljastaa laatikoiden sisältöä zoomaamatta.

5.2.1 Semanttinen zoomaus

Semanttinen zoomaus toteutetaan siten, että laatikko paljastaa sisältönsä kameran tullessa tarpeeksi lähelle. Laatikot reagoivat kameran sijaintiin kameravyöhykkeiden mukaisesti. Mitä matalampi laatikon hierarkiataso on, sitä kauemmaksi laatikosta vyöhyke ulottuu. Samalla hierarkiatasolla olevien säiliöiden vyöhykkeet ulottuvat yhtä kauas. Kuvassa 18 on havainnollistettu kameravyöhykkeitä suhteessa laatikoiden hierarkiatasoihin. Tiedostolaatikoilla vyöhykkeet ovat niin pienet, ettei kamera pysty olemaan useamman vyöhykkeen sisällä samaan aikaan. Tähän päädyttiin, sillä liian monen sisänäkymän yhtäaikainen näyttö toi liikaa yksityiskohtia ja muutti näkymän sekavan oloiseksi.

Tiedostolaatikko muuttuu läpinäkyväksi kameran ollessa vyöhykkeen sisällä ja vastaavasti kiinteän väriseksi kameran poistuessa. Muutos tehdään *triggered*-signaalin



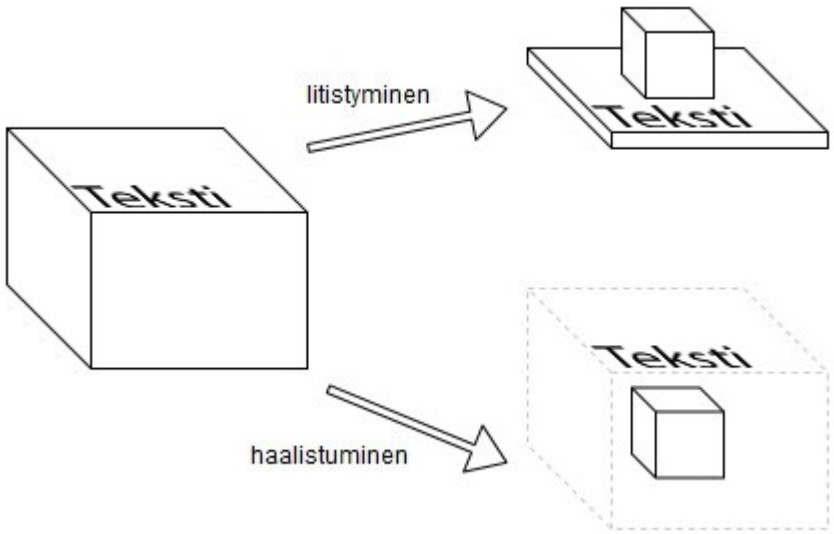
Kuva 18: Matalamman hierarkiatason omaavan laatikon kameravyöhyke ulottuu kauemmas laatikosta kuin korkeammalla hierarkialla olevan laatikon. Kuvassa kameran katsotaan olevan sinisen ja oikean puoleisen punaisen laatikon vyöhykkeellä.

ja kameran *positionChanged*-signaalin avulla. Laatikko kuuntelee kameran signaalia ja tarkistaa signaalin mukana saadun kameran sijainnin avulla, onko kamera vyöhykkeen sisällä ja tallentaa siitä tiedon. Tiedon ja *triggered*-signaalin avulla tarkistetaan kameran suhde laatikkoon jokaisen kuvan piirron yhteydessä ja läpinäkyvyysarvoa päivitetään sen mukaisesti. Muutos on kytketty *triggered*-signaalin mukana saatuun delta-aikaan, jonka avulla läpinäkyvyyttä muutetaan hiljalleen.

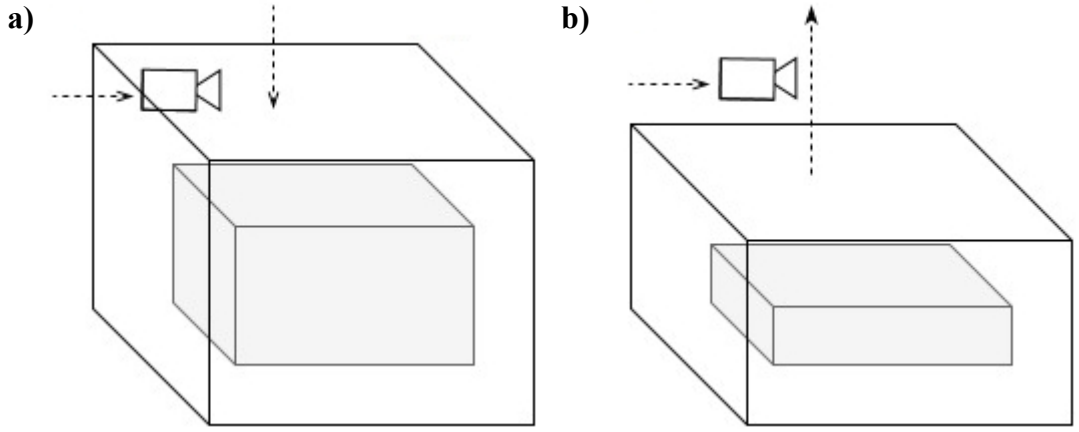
Koska läpinäkyvyys poistettiin säiliölaatikoilta, toteutettiin sisällönpaljastaminen säiliölaatikoiden yhteydessä litistymisen avulla. Kameran tullessa vyöhykkeen sisään, laatikko litistyy alustaksi sisällään oleville laatikoille. Vastaavasti kameran siirtyessä kauemmaksi, laatikko paisuu alkuperäiseen kokoonsa. Litistyminen käsitellään läpinäkyvyyden tapaan *positionChanged*- ja *triggered*-signaalien avulla.

Litistymisen käyttö auttoi myös ratkaisemaan laatikon nimen sijoittamisongelman. Tilannetta on havainnollistettu kuvassa 19. Jos säiliöt olisivat haalistuneet, teksti olisi jäänyt joko leijumaan paikoilleen näkymättömän laatikon päälle tai se olisi pitänyt haalistaa pois näkyvistä laatikon mukana. Ensimmäisessä tilanteessa ongelmana olisi se, että teksti olisi haitannut sisempien rakenteiden näkyvyyttä ja se tekisi sotkuisin vaikutelman. Toisessa tilanteessa ongelmana olisi se, että käyttäjän olisi pitänyt poistua aina vyöhykkeen sisältä nähdäkseen laatikon nimen. Kun laatikko litistyy, teksti voi seurata laatikon mukana.

Kameravyöhykkeet määritetään aina tietylle etäisyydelle laatikon reunoista. Tätä varten *QCuboidMeshistä* periytettiin uusi luokka, *CuboidMesh*, joka pitää tallessa laatikon mitat pullistuneessa muodossa. Tällöin saadaan vyöhykkeen yläraja määritettyä täyden



Kuva 19: Läpinäkyvyyden korvaaminen litistymisellä auttoi löytämään laatikon tekstille selkeän paikan. Siirtämällä tekstin laatikon reunan mukana alas saadaan kokonaiskuva pysymään selkeämpänä kuin jos teksti jäisi paikoilleen haalistuneen laatikon reunalle.

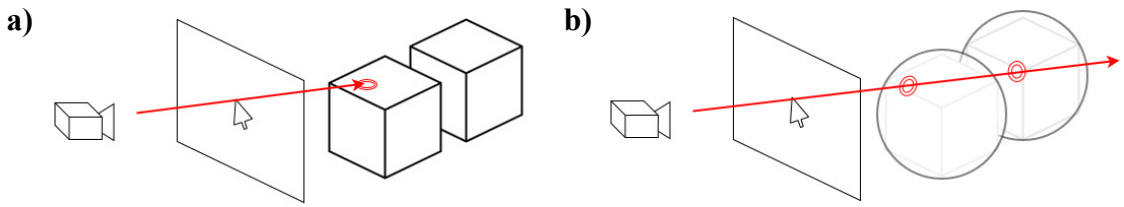


Kuva 20: Jos kameravyöhyke määräytyy laatikon sen hetkisen muodon mukaisesti, kameran havainnoimisessa on ongelma: (a) kameran tulo vyöhykkeeseen saa laatikon litistymään, (b) mutta laatikon litistyminen saa kameran poistumaan vyöhykkeestä, jolloin laatikko nousee.

koon mukaisesti. Jos vyöhykkeet määräytyisivät laatikon sen hetkisen muodon mukaisesti, saattaisi vyöhykkeen sisällä liikkuva kamera poistua vyöhykkeen sisältä laatikon kutistuessa. Tätä on havainnollistettu kuvassa 20. Tällöin tarkastelussa huomattaisiin kameran poistuminen vyöhykkeestä, jolloin laatikko lähtisi paisumaan takaisin muotoonsa ja kamera havaittaisiin jälleen vyöhykkeen sisällä jättäen laatikon nykivään tilaan. Kun vyöhykkeet määräytyvät täyden muodon mukaisesti, pysyy kamera vyöhykkeen sisällä, vaikka näkyvä laatikko kutistuisi, eikä nykimistä tapahdu.

5.2.2 Interaktiivisuus

Käyttäjälle tarjotaan myös mahdollisuus paljastaa lisää sisältöä ilman lähelle zoomausta manipuloimalla laatikoita suoraan osoittamalla tai klikkaamalla niitä. Hiiren tapahtumien tunnistuksessa käytetään olionpoimija-komponenttia, *QObjectPickeriä*, joka kertoo sen omistamaan entiteettiin kohdistuvista tapahtumista. Poimintametodi määritetään ikkunan *QRenderSettings*-oliolle, joten poimintametodi on yhtenäinen kaikille maiseman entiteeteille: joko poiminta tapahtuu siten, että kaikki kursorin kohdalla olevat entiteetit saavat ilmoituksen hiiren syötteestä tai vain ensimmäinen, lähimpänä oleva entiteetti. Vastaavasti poiminta-alue voidaan määrittää vastaamaan joko entiteetin monikulmioverkkoa tai sen ympärille muodostettua pallomaista raja-aluetta, joka pitää monikulmioverkon sisällään. Kuvassa 21 on havainnollistettu poimintametodin ja poiminta-alueiden eroja. Tässä yhteydessä on oleellista, että poiminta rekisteröityy sille entiteetille, jonka päällä kursori oikeasti on, joten siksi käytetään kuvan 21a mukaista asetusta poiminnassa.



Kuva 21: Olionpoimijan poimintatapoja ovat esimerkiksi a) lähin poiminta monikulmioverkon mukaisesti ja b) kaikki poiminnat raja-alueen mukaisesti.

Säiliölaatikon sisältöä voi tarkistella klikkaamalla laatikko litteäksi. Vastaavasti sisällön saa piilotettua klikkaamalla laatikkoa uudelleen. Tieto klikkauksesta saadaan olionpoimijan *clicked*-signaalin avulla. Laatikko määrittää nykyisen tilansa mukaisesti alkaako se litistyä vai paisua: jos laatikko on paisunut, se litistyy ja litistynyt laatikko puolestaan alkaa paisumaan vain, jos kamera ei ole sen vaikutusalueen sisällä.

Sen sijaan klikkaamalla tiedostolaatikkoa käyttäjä voi automaattisesti siirtyä laatikon sisälle editoritilaan. Saatuaan tiedon klikkauksesta, laatikko lähettää siitä signaalin kameraohjaimelle. Ohjain siirtää kameraa automaattisesti suorinta polkua pitkin laatikkoon, jolloin käyttäjä pystyy seuraamaan sijaintinsa ja orientaationsa muutosta. Kun kamera on saavuttanut pisteen, jossa laatikko huomaa kameran olevan sisällään, siirtyy kamera katsomaan sisänäkymää edellisessä luvussa esitetyllä tavalla.

Käyttäjä voi myös kurkistaa tiedostolaatikon sisään osoittamalla sitä nähdäkseen sisänäkymän sisällön. Tämä itsessään on ristiriidassa sen kanssa, että säiliölaatikon sisällön voi paljastaa klikkaamalla. Loogisempaa olisi, jos molemmissa tapauksissa käytettäisiin samaa hiiren tapahtumaa. Tiedostolaatikkoon siirtyminen osoituksen avulla kuitenkin häiritsisi muuta navigaatiota. Esimerkiksi muita laatikoita olisi hankala klikkailla, jos täytyisi jatkuvasti vältellä matkan varrella olevia tiedostolaatikoita pitääkseen kameran paikoillaan. Toisaalta säiliölaatikoiden osoittelussa olisi hankalaa hallita paljastettujen kerrosten määrää. Kun käyttäjä osoittaisi yhtä laatikkoa, litistyisi päälimmäisen laatikon lisäksi myös loput kursorin kohdalla olevat laatikot, jolloin sisältöä paljastuisi liikaa. Lisäksi osoittamisessa tulisi samankaltainen ongelma kuin kuvassa 20: laatikon litistyessä kursori ei enää osoittaisikaan laatikkoa, jolloin laatikko paisuisi. Siten päädyttiin lopulta käsittelemään sisällön paljastukset eri tapahtumilla.

Tieto osoittamisesta saadaan olionpoimijalta *entered*- ja *exited*-signaaleilla. Oletusarvoisesti olionpoimija rekisteröi vain painallukset, joten signaalien saamiseksi olionpoimijalle täytyy erikseen määrittää osoituksen kuuntelu. Laatikko tietää olevansa osoitettavana, jos se on saanut e*ntered*-signaalin, mutta ei e*xited*-signaalia. Osoitus käsitellään jälleen *QFrameActionin* avulla animoiden laatikkoa läpinäkyvämmäksi, jos laatikkoa osoitetaan. Muutoin laatikko animoidaan kiinteän väriseksi.

Olionpoimija ei ainakaan monikulmioverkkopoiminnassa toimi aivan saumattomasti. Poimijasta saatiin sisääntulo- ja poistumissignaaleja, vaikka kursori selvästi pysyi saman laatikon päällä. Tällöin värimuutoksessa tapahtui ikävää välkkymistä, kun hiirtä raahasi laatikon yli. Tämä ratkaistiin lisäämällä signaaleihin reagoimiseen viive *QTimer*-ajastimen avulla. Ajastimelle asetetaan aika, jonka kuluttua ajastuksen aloituksesta saadaan *timeout*-signaali. Kun uusi poimintasignaali rekisteröidään entiteetillä, tallennetaan se ja aloitetaan ajastus. Tässä vaiheessa poimintasignaaliin ei vielä reagoida. Jos entiteetille saadaan uusi signaali, otetaan se jälleen talteen ja ajastus aloitetaan alusta. Kun ajastimelta saadaan *timeout*, eli kun uusia signaaleja ei ole hetkeen tullut, reagoidaan viimeksi saadun signaalin mukaisesti. Ajastimen odotusaikana käytettiin 50 millisekuntia, jolla saa nopean välkynnän karsittua pois, mutta reagointi näyttää kuitenkin käyttäjälle lähes viiveettömältä.

Säiliölaatikoiden osoitus käsitellään samaan tapaan kuin tiedostolaatikoiden, mutta läpinäkyvyyden sijasta laatikon väriä muutetaan kirkkaammaksi. Näin käyttäjän on helpompi hahmottaa minkä laatikon kanssa hän on kanssakäymisessä. Säiliölaatikon väriä muutetaan myös kirkkaammaksi, jos sen päällä on käyttäjän litistämä laatikko. Tällä indikoidaan se, että päällä litistetty laatikko estää alemman pullistumisen oli käyttäjä sitten alemman laatikon kameravyöhykkeen sisällä tai ei. Näin ei häiritä käyttäjää tarkistelemasta sisältöä.

Myös klikkaus- ja osoitussignaalien yhtäaikainen käsittely tuottaa hankaluuksia. Jos esimerkiksi klikkauksen aikana liikuttaa kameraa siten, että klikkaus alkaa yhden laatikon päällä ja loppuu toisen laatikon päällä, alkaa *entered*-signaali toimimaan väärin. Signaali ei rekisteröidykään osoitettavalle laatikolle vaan sille, jonka päältä klikkaus alkoi. Ongelman saa korjattua klikkaamalla johonkin osaa ikkunaa siten, että kursoria ei raahaa laatikosta toiseen.

5.3 Tekstieditori

Alkuperäisessä prototyypissä tiedostolaatikon sisällä olleet kuvakaappaukset korvaattiin tekstieditorilla. Editorin tarkoituksena on havainnollistaa prototyypin toimintaa siten, että 3D-näkymässä voidaan tehdä muokkausta navigoinnin lisäksi. Toteutukseen käytettiin Qt 3D:stä löytyvää *Scene2D*-moduulia, jolla voidaan piirtää käyttöliittymä osaksi entiteetin materiaalia. *Scene2D* tukee vain QML-käyttöliittymiä, joten editorin kohdalla jouduttiin poikkeamaan C++:n käytöstä, sillä vastaavaa moduulia ei perinteisille Qt:n widgeteille löydy.

Editorille toteutettiin yksinkertaisen tekstinmuokkausohjelman toiminnot: kursorin siirtäminen tekstissä, tekstin muokkaus ja dokumentin tallennus. Pääsääntöisesti tekstin

käsittely tapahtuu näppäimistöllä ja editori tukee yleisimpiä pikakomentoja tekstin kopionnille, liittämiselle, leikkaukselle, kumoamiselle ja tallennukselle. Tallennus tapahtuu myös automaattisesti laatikosta poistuttaessa, jotta käyttäjä voi nopeasti liikkua laatikoiden välillä tehden muutoksia.

Editorin käyttöliittymä tehtiin käyttämällä Qt:n valmiita QML-komponentteja. Editorin rakenne on esitettynä ohjelmassa 4. Käyttöliittymä koostuu vieritysalueesta *ScrollViewstä* ja sen sisällä olevasta tekstialueesta *TextAreasta*. Dokumenttikäsittelijä, *DocumentHandler*, on lainattu Qt:n tekstieditoriesimerkistä [22]. Sekä hiiren että näppäimistön syötteet saadaan 3D-näkymän entiteettien kautta.

```
1   Item {
2     id: root
3     width: 1024
4     height: 1024
5     enabled: true
6     property bool handleInput: false
7     property string filename: ""
8
9     Connections {
10      target: innerViewHandler
...     /* sisänäkymältä saatujen signaalien käsittelyt */
24    }
25    Connections {
26      target: keyboardHandler
...     /* sisänäkymän näpääinkäsittelijältä saatujen signaalien käsittelyt */
118   }
119   ScrollView {
120      anchors.fill: parent
121      x: 0
122      y: 0
123      width: parent.width
124      height: parent.height
125      ScrollBar.vertical.policy: ScrollBar.AlwaysOn
126      TextArea {
127        id: textArea
128        cursorVisible: true
129        selectByMouse: true
130        textFormat: Qt.PlainText
131        wrapMode: TextEdit.Wrap
...        /* tekstin tyylimäärittelyt */
134        background: Rectangle {
...          /* taustan tyylimäärittelyt */
140        }
141      }
142    }
143    DocumentHandler {
...      /* tiedostonlatauksen käsittelyt */
155    }
156 }
```

Ohjelma 4: Tekstieditorin käyttöliittymä koostuu vieritys- ja tekstialueesta.

5.3.1 Editorin liittäminen 3D-näkymään

Jotta editorin saa osaksi 3D-näkymää, luodaan sitä varten QML-materiaali, *QmlMaterial*, joka rakennetaan ohjelmassa 5. *Scene2D* ja sen käyttämä *QRenderTargetOutput* liitetään osaksi kehysgraafia. Piirtoa varten luodaan ulostulotekstuuri ja *Scene2D*:lle määritetään sen käyttämän QML-näkymän juurielementti, joka on ohjelman 4 rivin 1 *Item*. Juurielementin sisältö piirretään ulostulotekstuuriin. Rivillä 26 ulostulotekstuuri asetetaan osaksi materiaalia, jolloin se saadaan 3D-näkymään asettamalla materiaali sisänäkymän komponentiksi.

Lisäksi *Scene2D*:lle määritetään, sallitaanko hiiren kuuntelu. Jos sallitaan, hiiren syötteet rekisteröidään QML-näkymässä. Hiiren syötteet saadaan sen sisänäkymän kautta, jolla materiaali on käytössä. Tämä vaatii myös olionpoimijan lisäämisen sisänäkymälle. Tekstin maalaamista varten olionpoimijalle pitää erikseen määrittää myös vetämisen salliminen.

QML-materiaalin voi piirtää kerrallaan vain rajoitetulle määrälle entiteettejä. Kehitysympäristössä noin 12 QML-materiaalin yhtäaikainen käyttö kaataa ohjelman. Tarkkaa syytä kaatumiselle ei tiedetä, mutta ongelma saattaa liittyä siihen, että materiaali on *Scene2D*:n kautta osa kehysgraafia, jolloin ulostulotekstuureja päivitetään

```
1  QmlMaterial::QmlMaterial( QString const& textureSource, const QSize screenSize,
2    Qt3DRender::QFrameGraphNode* frameGraphNode, QNode* parent ):
3    Qt3DExtras::QTextureMaterial( parent ),
4    qmlView_( new QQuickView() ),
5    texture_( new Qt3DRender::QTexture2D( this ) ),
6    qmlTextureRenderer_( new Qt3DRender::Quick::QScene2D( frameGraphNode ) )
7  {
8    qmlView_->setSource( QUrl( textureSource ) ); // textureSource = QML-kooditiedosto
9
10   texture_->setSize( screenSize.width(), screenSize.height() );
11   texture_->setFormat( Qt3DRender::QAbstractTexture::RGBA8_UNorm );
12   texture_->setGenerateMipMaps( true );
13   texture_->setMagnificationFilter( Qt3DRender::QAbstractTexture::Linear );
14   texture_->setMinificationFilter( Qt3DRender::QAbstractTexture::LinearMipMapLinear );
15   texture_->setWrapMode(
16     Qt3DRender::QtextureWrapMode( Qt3DRender::QTextureWrapMode::ClampToEdge, texture_ ) );
17
18   qmlRenderTargetOutput_ = new Qt3DRender::QRenderTargetOutput( qmlTextureRenderer_ );
19   qmlRenderTargetOutput_->setAttachmentPoint( Qt3DRender::QRenderTargetOutput::Color0 );
20   qmlRenderTargetOutput_->setTexture( texture_ );
21
22   qmlTextureRenderer_->setOutput( qmlRenderTargetOutput_ );
23   qmlTextureRenderer_->setItem( qmlView_->rootObject() );
24   qmlTextureRenderer_->setMouseEnabled( true );
25
26   setTexture( texture_ );
27 }
```

Ohjelma 5: QML-näkymän piirto osaksi materiaalia tapahtuu Scene2D:n avulla.

jokaisella kuvalla. Koska ohjelmarakennetta visualisoidessa tarvittavien sisänäkymien määrä todennäköisesti on suurempi kuin 12, keskitettiin materiaalin luonti ja käyttö sisänäkymäkäsittelijälle. Se huolehtii kaikesta QML-näkymän ja 3D-näkymän välillä tapahtuvasta kommunikaatiosta.

Mikäli QML-tekstialuetta käytettäisiin perinteiseen tapaan ilman 3D-moduulia, ei näppäimistön syötteitä tarvitsisi erikseen käsitellä vaan ne rekisteröityisivät automaattiseesti tekstialueelle, jos alue on käyttöliittymässä aktiivinen elementti. Pikakomentoja voitaisiin QML-koodissa ottaa käyttöön esimerkiksi *Shortcutin* avulla oheisella tavalla:

```
Shortcut {
    sequence: StandardKey.Paste
    onActivated: textArea.paste()
}
```

Kun QML-näkymä on käytössä *Scene2D*:n kautta, näppäinsyötteet eivät rekisteröidy tekstialueelle eivätkä myöskään esimerkiksi Shortcutit toimi. Syötteet täytyy tuoda erikseen jonkin 3D-näkymässä olevan entiteetin kautta ja käsitellä QML-koodissa yksittäisen kirjaimen lisäystä myöten. Syötteistä vastaava entiteetti on sisänäkymäkäsittelijä ja QML-näkymään ne saadaan sen *QKeyboardHandler*-komponentin avulla. Komponentin *pressed*-signaalia kuunnellaan QML-koodissa ja sen mukana saadun tapahtuman sisältämien näppäinsyötteiden perusteella tapahtuma käsitellään tekstialueen metodeilla ohjelman 6 mukaisesti.

Koska käsittely pitää erikseen tehdä kaikille halutuille toiminnoille ja editori korvataan myöhemmissä toteutuksissa jollakin muulla ohjelmalla, tuetaan siinä vain oleellisimpia toimintoja näppäimillä, joilla saadaan demoamistarpeisiin riittävä toiminnallisuus:

- Nuolinäppäin kursorin liikuttamiselle
- Shift + nuolinäppäin tekstin valinnalle
- Ctrl + A koko tekstin valinnalle

```
25   Connections {
26       target: keyboardHandler
27       onPressed: {
28           if( !root.handleInput ) {
29               event.accepted = true
30               return
31           }
32           if( event.modifiers === Qt.ControlModifier ) {
33               switch ( event.key ) {
34               case 'C'.charCodeAt(0):
35                   textArea.copy()
36                   break
...                ...
```

Ohjelma 6: Näppäinsyötteet on erikseen käsiteltävä QML-koodissa entiteetin näppäinkäsittelijältä saadun tapahtuman mukaisesti.

- Ctrl + C, Ctrl + V ja Ctrl + X kopioinnille, liittämiselle ja leikkaamiselle
- Ctrl + Z ja Ctrl + Y kumoamiselle ja uudelleen tekemiselle
- Ctrl + S tiedoston tallennukselle
- Delete- ja Backspace tekstin poistolle

Sisänäkymäkäsittelijä huolehtii QML-materiaalin luonnin yhteydessä siitä, että materiaalin resoluution on sama kuin ikkunalla. Sekä juurielementin että ulostulotekstuurin koko asetetaan samaksi ja koko määrittyy käytössä olevan ikkunan mukaan, jotta käyttäjän siirtyessä editorinäkymään editorin sisältö piirtyy tarkkana. Jatkossa sisänäkymäkäsittelijä huolehtii sisänäkymän materiaalien vaihdosta sen mukaan onko kamera tiedostolaatikossa: kameran ollessa laatikon sisällä käytetään sen sisänäkymällä QML-materiaalia ja kameran ollessa ulkopuolella käytetään kuvakaappausmateriaalia, *ImageMaterialia*.

5.3.2 Editorin käyttö

Käyttäjän tullessa editoritilaan, sisänäkymäkäsittelijä saa tiedon tapahtumasta laatikon lähettämästä signaalista. Signaalin mukana saadaan osoitin laatikon sisänäkymään, jolta käsittelijä pyytää tiedostonimen sisänäkymän tekstidokumenttiin. Sisänäkymäkäsittelijä lähettää *loadViewContent*-signaalin mukana tiedostonimen, jonka avulla dokumenttikäsittelijä lataa tiedoston sisällön tekstialueeseen. Ohjelmassa 7 esitetään ohjelmassa 4 näkyvien sisänäkymäkäsittelijän ja dokumenttikäsittelijän signaalienkäsittelyt. Materiaali asetetaan sisänäkymän käyttöön ja sisänäkymä lisätään *Scene2D*:n kuuntelemiin entiteetteihin, jolloin sisänäkymän olionpoimijan rekisteröimät hiiren tapahtumat kirjautuvat vastaavaan kohtaan tekstieditorin näkymää.

Halutessaan poistua editoritilasta poistumispyyntö havaitaan kameraohjaimessa, joka lähettää siitä signaalin sisänäkymäkäsittelijälle. Sisänäkymäkäsittelijä kieltää QML-näkymältä näppäimistön syötteiden käsittelyn ja pyytää dokumenttikäsittelijää tallentamaan dokumentin sisällön. Tämän jälkeen sisänäkymäkäsittelijä pyytää kehysgraafin *QRenderCapturelta* kuvakaappausta kameran näkymästä. Kuvakaappauksen ajaksi kamera lukitaan laatikon sisään, jotta kuvan saa koko ruudun täyttävästä editorista. Kuva tallennetaan levylle ja päivitetään tekstuuriksi sisänäkymän omaan kuvakaappausmateriaaliin, joka vaihdetaan QML-materiaalin tilalle. Lopuksi sisänäkymä poistetaan *Scene2D*:n kuuntelemista entiteeteistä ja kameraohjaimelle lähetetään signaali tallennusprosessin valmistumisesta, jolloin kamera pääsee poistumaan laatikosta.

```
9    Connections {
10     target: innerViewHandler
12     onLoadViewContent: {
13       root.filename = filename
14       document.load( filename )
15     }
17     onSaveViewContent: {
18       document.saveAs( filename )
19     }
21     onEnableInput: {
22       root.handleInput = inputEnabled
23     }
24   }
...
143  DocumentHandler {
145    id: document
146    document: textArea.textDocument
147    cursorPosition: textArea.cursorPosition
150    selectionStart: textArea.selectionStart
151    selectionEnd: textArea.selectionEnd
153    onLoaded: {
154      textArea.text = text
155      textArea.cursorPosition = textArea.length
156    }
158    onError: {
159      textArea.text = message
160    }
151  }
```

Ohjelma 7: Dokumenttikäsittelijä lataa sisänäkymäkäsittelijältä saadun tiedostonimen perusteella dokumentin sisällön tekstialueeseen.

Kuvakaappauksen huonona puolena on se, että kuva on tallennettava huomattavasti pienemmän kokoisena kuin minä se saadaan *QRenderCapturelta*. Tämä johtuu siitä, että *QRenderCapturelta* saatu *QImage* pidetään ohjelman muistissa, koska se on osa materiaalia. Liian suurella kuvakoolla ohjelma kaatuu virheeseen muistin loppumisesta. Esimerkiksi kehitysympäristössä käytetty näytön resoluutio 1920 * 1200 sai muistiongelman aikaiseksi, kun kuvakaappauksia otti noin kymmenestä sisänäkymästä. Kun kuvat tallentaa 960 * 600 resoluutiolla, vastaavaa virhettä ei tule ja kuvien määrä vaikuttaisi olevan rajaton. Lopputuloksena kuvankaappausmateriaalissa kuvan yksityiskohdat näkyvät sotkuisina.

6 ARVIOINTI

Tässä luvussa arvioidaan tavoitteiden toteutumista alkuperäisten ja visualisoinnin yhteydessä todettujen vaatimusten osalta sekä Qt 3D:n soveltumista prototyypin toteutukseen. Lisäksi pohditaan jatkokehitysideoita nykyiseen toiminnallisuuteen ja integrointia Kactus2:een.

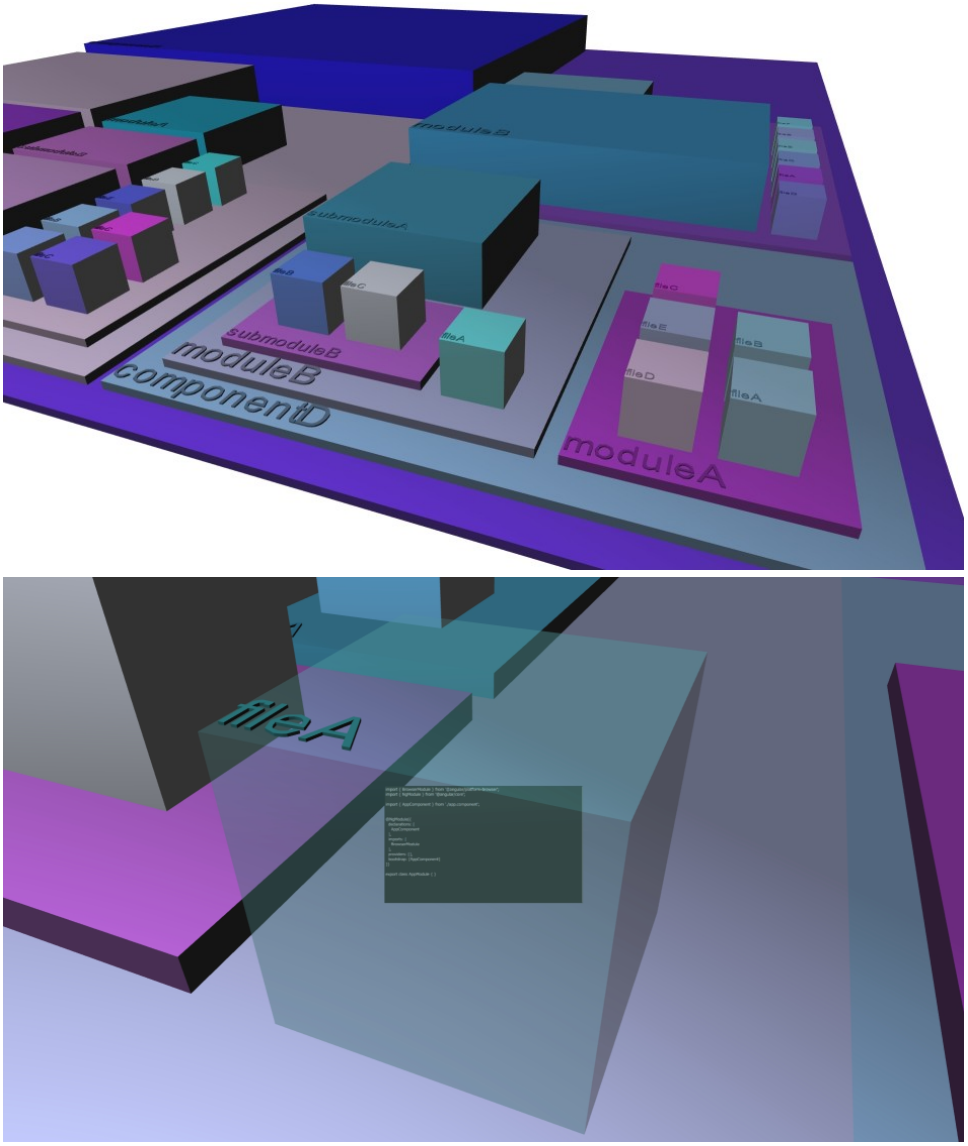
6.1 Tavoitteiden toteutuminen

Tämän diplomityön tavoitteena oli toteuttaa prototyyppi, jossa voidaan siirtyä sulavasti kolmi- ja kaksiulotteisten näkymien välillä. Prototyypissä haluttiin visualisoida ohjelman komponenttien hierarkia 3D-maailmana ja komponenttien yksityiskohdat 2D-näkyminä. Visualisoinnilla haluttiin kuormittaa käyttäjää mahdollisimman vähän ja estää eksyminen navigoinnin yhteydessä. Lisäksi haluttiin yhdistää yksityiskohtien muokkaus osaksi visualisointia. Kaiken kaikkiaan siirtymä eri visualisointitapojen välillä saatiin sulavaksi molempiin suuntiin. Siirtymäsekvenssi 3D:stä 2D:hen on esitetty kuvassa 22.

Tavoitteen mukaisesti, hierarkiarakenne kuvataan 3D-objekteina. Alkuperäisestä suunnitelmasta poiketen, läpinäkyvyyttä ei voitu käyttää muualla kuin sisimmissä rakenteissa. Läpinäkyvyyden korvaava toiminta, litistyminen, toimii kuitenkin tehtävässä jopa paremmin kuin läpinäkyvyys, sillä komponenttien nimikyltit saatiin aseteltua siistimmin näkymään.

3D:n vuoksi toteutettiin navigointi siten, että käyttäjä voi vapaasti tutkia maailmaa eri kuvakulmista, mutta turhat liikkeet, kuten ylösalaisin kääntyminen ja rakenteiden alapuolelle siirtyminen rajattiin pois eksymisen ehkäisemiseksi. Navigointia helpotetaan luomalla maamerkkejä värien ja tekstin sijainnin avulla. Lisäksi käyttäjän liike automatisoidaan tilanteissa, joissa liike on rajattua, esimerkiksi tiedostolaatikon sisällä tai aloittaessa navigointia alusta. Liikkuminen ja kääntyminen tapahtuvat sulavasti, jolloin käyttäjän on helpompi hahmottaa sijaintinsa ja orientaationsa muutosta.

Käyttäjä voi itse hallita tarkastelutasoaan semanttisen zoomauksen avulla. Näkyvyyttä on rajoitettu siten, että eri näkymätasoilla käyttäjälle näkyy vain oleellinen tieto, jolloin näkymän sisältö kuormittaa käyttäjää mahdollisimman vähän. Lisäksi käyttäjän on




```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';


@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})

export class AppModule { }
```

Kuva 22: Siirtymä 3D-valikosta halutun tekstitiedoston editoritilaan.

mahdollista paljastaa lisää sisältöä halutessaan klikkailemmalla ja osoittamalla. Interaktiota laatikoiden kanssa korostetaan animaatioiden avulla.

Kaksiulotteinen sisänäkymä haluttiin toteuttaa siten, että näkymässä pystyy muokkaamaan tekstimuotoista dokumenttia. Editoritilassa dokumenttia voi muokata samalla tapaa kuin perinteisessä tekstieditorissa: hiirellä ja näppäimillä voi liikuttaa kursoria sekä muuttaa valittua tekstiä. Lisäksi tekstiä voi kirjoittaaa, kopioida, liittää ja leikata. Editorin toiminnallisuus on tosin yleisiin tekstieditoreihin verrattuna jopa varsin puutteellinen. Syötekäsittely piti tehdä pitkälti itse, eikä sen hiominen pitkälle ollut tässä tapauksessa perusteltua, koska editori korvataan myöhemmin toisenlaisella työkalulla. Sen avulla voidaan kuitenkin hyvin demonstroida, että prototyyppissä yhdistyy visualisointi ja muokkaus.

Yllättävää oli, että varsin yksinkertaisen näkymän toteutus on kuitenkin monimutkainen. Käytännössä maisema koostuu vain paikoillaan olevista laatikoista ja paikoillaan kääntyilevistä tasoista. Visualisoinnin kannalta huomioon otettavat asiat, kuten navigointi ja tapahtumien indikoiminen käyttäjälle, lisäävät heti toteutuksen monimutkaisuutta. Lisäksi tekninen puoli, kuten piirtoprosessin toiminta alla ja kehitysympäristön omat oikut, tekee kokonaisuudesta entistäkin haastavampaa. Koska vastaavaa järjestelmää ei ole ennen tehty, ei myöskään ollut niin sanottua valmista tarttumapistettä, josta olisi voinut verrata hyviä ja huonoja ratkaisuja sekä tehdä omia ratkaisuja niiden pohjalta.

6.2 Qt 3D:n soveltuminen työhön

Qt 3D:n suurin hyöty on siinä, että se tekee paljon asioita käyttäjän puolesta. Kuten läpinäkyvyyden yhteydessä voitiin todeta, piirron yhteydessä tehdään oletuksia: Koska käyttäjä tulee todennäköisesti tarvitsemaan syvyystestiä, se on oletusarvoisesti käytössä. Syvyystestin ohella käytetään oletuksena myös takapintojen poistoja ja sallitaan puskuriin kirjoituksessa kaikkien värikanavien käyttö [19]. Tällainen asetelma on yleisesti käytössä OpenGL-ohjelmissa, joten käyttäjä saa helposti renderöityä näkymän tutustumatta varsinaiseen OpenGL:n piirtoliukuhihnaan. Lisäksi Qt 3D:sta löytyy paljon hyödyllisiä komponentteja. Tässä työssä käytetyistä komponenteista suurin osa oli valmiita komponentteja ja kolme omaa komponenttia toteutettiin periyttämällä ne Qt 3D:n omista komponenteista.

Qt 3D:n suurin ongelma on sen dokumentaation puute. Luokkien metodit ovat suurimmalta osin hyvin pinnallisesti dokumentoitu ja paikoittain dokumentaatio puuttuu tyystin. Vaikka Qt 3D tekee paljon asioita käyttäjän puolesta ja Qt:n sivuilta löytyy esimerkkejä hyvin yksinkertaisiin käyttötapauksiin, etenkin C++-esimerkkien

tapauksessa, tilanne muuttuu hankalaksi heti kun käyttäjän täytyy ottaa kantaa piirtotekniikkaan ja muihin yksityiskohtiin. Käyttö on hankalaa etenkin, jos tietokonegrafiikka ja OpenGL eivät ole ennestään tuttuja.

Edellisestä johtuen, tämän työn aikana suurin osa ongelmanratkonnasta tapahtui tarkastelemalla Qt 3D:n lähdekoodia ja testitapauksia. Mikäli kyseessä ei olisi avoimen lähdekoodin kehitysympäristö, ei tätä diplomityötä olisi varmaan kyseisellä ympäristöllä tehty – tai lopputuloksena olisi ollut toiminnaltaan huomattavasti laihempi järjestelmä. Kynnys Qt 3D:n käyttöön on varmasti vielä suurempi projekteissa, joissa on tiukat aikarajoitteet. Todennäköisesti jatkokehityksesssä Qt 3D on silti paras vaihtoehto, sillä siihen lisätään vielä uusia ominaisuuksia ja toivon mukaan dokumentaatiokin seuraa pian perässä.

6.3 Jatkokehitys

Tässä luvussa käsitellään prototyypin nykyisen toiminnallisuuden parannusehdotuksia ja integrointia Kactus2:een.

6.3.1 Parannusehdotuksia

Prototyyppi sellaisenaan ei riitä juuri muuhun kuin tuntuman hakemiseen varsinaisesta järjestelmästä. Vaikka toiminnallisuutta ei paljoa vielä ole, on siinäkin hiottavaa. Ennen kaikkea prototyypin olisi pystyttävä toimimaan erilaisissa käyttöympäristöissä. Kun prototyyppiä testattiin eri laitteistoilla, ei sitä kaikilla koneilla saatu edes käyntiin. Joillakin koneilla ohjelma saatiin toimimaan määrittämällä käyttöön eri OpenGL:n versio. Kuten useimmissa ohjelmistoissa, tämänkin yhteydessä olisi selvitettävä tarvittavat järjestelmän minimivaatimukset.

Laatikoiden nimet, etenkin pitkät sellaiset, näkyvät paikoin huonosti ja usein valikossa täytyy navigoida hyvin lähelle laatikkoa nähdäkseen tekstin. Tekstiä voisi suurentaa tai nostaa paremmin esille esimerkiksi laatikon osoituksen yhteydessä, jotta käyttäjän ei tarvitsisi siirtyä laatikon lähelle nähdäkseen sen nimen.

Tiedostolaatikot tai sisänäkymät voisivat pitää tallessa editorin tilaa, kuten kursorin paikkaa tai maalattua tekstin osaa, jolloin käyttäjän on helpompi palata keskeneräisen tehtävän pariin. Lisäksi tilojen ja dokumenttien tallennuksen voisi suorittaa sinä aikana, kun käyttäjä on jo navigoinut pois laatikosta, jolloin kameraa käyttäjineen ei tarvitsisi lukita laatikon sisälle tallennuksen ajaksi. Tämä vaatisi kuvakaappauksen nappaamisen esimerkiksi suoraan QML-materiaalin tekstuurin puskurista, jolloin *QRenderCapturesta* voisi luopua.

Prototyypin tekstieditori ei kuitenkaan riittäisi varsinaisen ohjelmistokehityksen tarpeisiin, vaan kehittäjät tarvitsevat työssään usein monimutkaisempia työkaluja. Jo työn alkuvaiheessa pohdittiin mahdollisuutta näyttää tiedostolaatikoiden sisällä näkymää varsinaisesta ohjelmistokehityksessä käytettävästä ohjelmasta kuten Visual Studiosta tai Qt Creatorista.

Lisäämällä erilaisia työkaluja, kuten tarttuma- ja kääntökäsiä, saataisiin samoille hiiren tapahtumille jaettua selkeämmin erillisiä toimintoja. Tällöin esimerkiksi olionpoiminnan yhteydessä havaittu klikkauksen ja osoituksen sekoittuminen voitaisiin poistaa, koska näkymän kääntäminen ja laatikon valinta tapahtuisivat erillisten työkalujen kautta eikä niiden käyttö yhtäaikaa olisi mahdollista. Lisäksi kameran kääntämisen sijasta voisi olla luontevampaa manipuloida objekteja suoraan esimerkiksi tarttumalla laatikkoon ja kääntäämällä sitä.

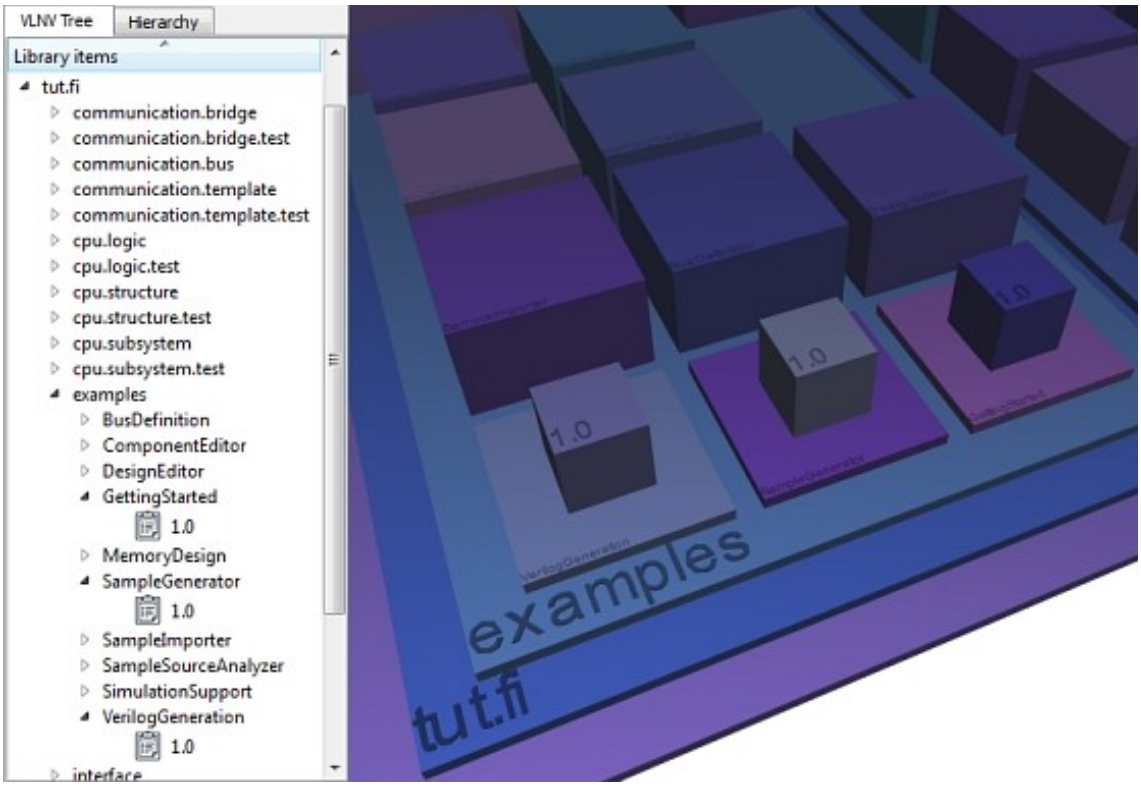
6.3.2 Integrointi Kactus2:een

Jo työn toteutuksen alusta asti on ollut selvillä, että prototyyppi integroidaan osaksi Kactus2:ta. 3D-valikkoa on tarkoitus käyttää IP-XACT-kirjaston komponenttien listaukseen ja sisänäkymissä puolestaan on tarkoitus näyttää komponentti Kactus2:n editorissa.

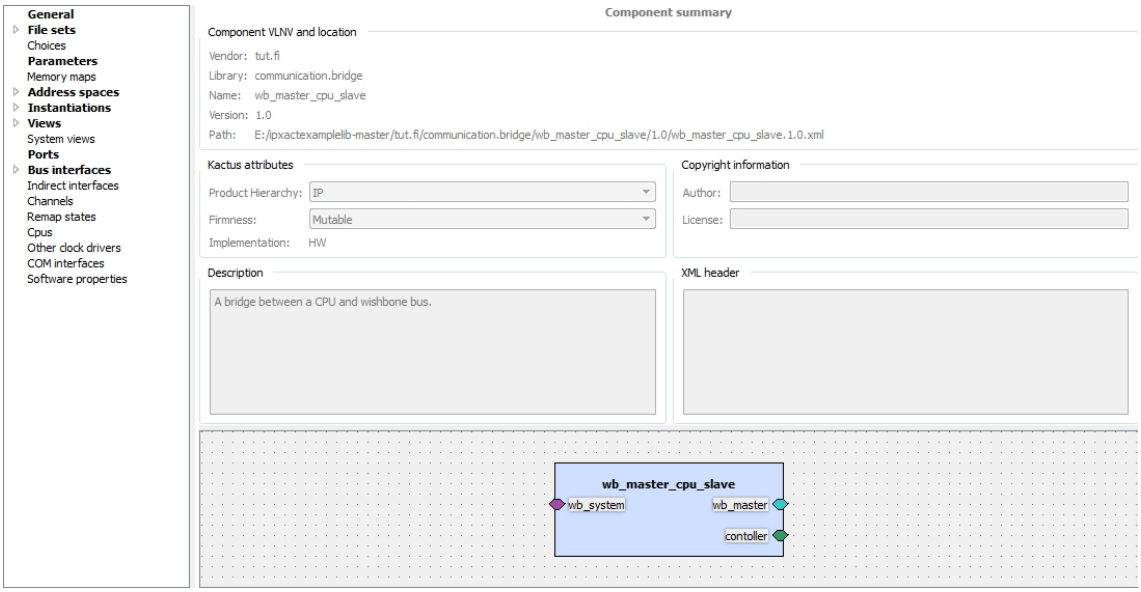
IP-XACT-komponenttien tunnistukseen käytetään nelitasoista VLNV-tunnistetta (Vendor-Library-Name-Version), jonka pohjalta Kactus2:ssa esitetään komponentit puumaisena hierarkiatasoihin pohjautuvana listana, VLNV-puuna. Kuvassa 23a vasemmalla on esitettynä erään kirjaston rakennetta avattuna VLNV-puussa. Esimerkki yhdestä VLNV-tunnisteesta on tut.fi (vendor) – examples (library) – GettingStarted (name) – 1.0 (version). Kuvassa oikealla on puolestaan sama puun rakenne syötettynä 3D-valikkoon. Kactus2:n valikossa ongelmana on listan kasvaminen kerrosten avaamisen yhteydessä. Mitä enemmän listassa on sisältöä, sitä hankalampi siinä on navigoida ja sen ulkonäkö vaihtelee myös sen mukaan mitkä tasot ovat auki. 3D-valikossa koko puolestaan ei muutu ja ulkonäkökin pysyy tutumpana, vaikka eri tasoja olisi aukaistu. Kuvassa 23b on puolestaan eräs komponentti muokattavana Kactus2:n editorissa, joka 3D-valikon yhteydessä esitettäisiin laatikon sisänäkymässä.

Kactus2:ssa on myös mahdollisuus hakea ja suodattaa näytettäviä komponentteja, joten näiden toimintojen tulokset pitäisi indikoida 3D-näkymässä esimerkiksi muuttamalla laatikoiden väriä haun yhteydessä sekä piilottamalla tai näyttämällä suodatuksen tulokset. Lisäksi laatikoissa käytettävien värien yhdistäminen komponenttien tyyppiin auttaisi nopealla silmäyksellä erottamaan erityyppiset komponentit toisistaan.

a)



b)



Kuva 23: a) Samat hierarkiatasot avattuna Kactus2:n VLNV-puussa ja prototyypin 3D-valikossa. b) Kactus2-integraation myötä tiedostolaatikon sisäkymässä näytettäisiin komponentti Kactus2:n editorissa.

7 YHTEENVETO

Visualisointi on tärkeä osa ohjelmistokehitystä, sillä kuva auttaa ymmärtämään monimutkaisiakin järjestelmiä sanallista selitystä tehokkaammin. Visualisoinnin kannalta on kuitenkin oleellista löytää tasapaino kuvan ilmaisuvoiman ja tehokkuuden välillä. Liika informaatio tekee kuvasta sekavan, mutta liian vähäinen tieto puolestaan ei riitä kuvaamaan järjestelmää tarpeeksi yksityiskohtaisesti. Tärkeää on löytää järjestelmästä oleellinen tieto, jonka avulla voidaan välittää sekä kokonaiskuva että tarpeelliset yksityiskohdat.

Suuren tietomäärään hallintaan on kehitetty erilaisia tekniikoita, joilla pyritään esittämään katsojalle oleellinen informaation ymmärrettävässä muodossa. Hyvänä ratkaisuna voidaan pitää semanttista zoomausta, jolla voidaan jakaa tieto eri karkeusasteilla ja zoomaustasosta riippuen näyttää tason suhteen oleellinen tieto. 3D:n avulla voidaan sitoa kuvaan suurempi informaatiomäärä, mutta vapaamman liikkuvuuden mukana tulee ongelmaksi käyttäjän eksyminen. Eksymistä voidaan kuitenkin ehkäistä rajatulla sekä automatisoidulla navigaatiolla. Animaatioilla, maamerkeillä ja sulavalla liikkeellä puolestaan autetaan käyttäjää hahmottamaan näkymässä tapahtuvia muutoksia.

Tässä työssä toteutettiin prototyyppi ohjelmasta, jolla voidaan yhtäaikaisesti visualisoida ohjelmakokonaisuuden komponenttien hierarkiarakennetta sekä muokata komponenttien yksityiskohtia. Prototyypilla on tarkoitus hakea tuntumaa uudenlaiseen järjestelmään, jolla ohjelmiston kehitystyötä pyritään nopeuttamaan navigoimalla ohjelmarakenteen sisällä siirtyen kokonaiskuvan visualisoinnin yhteydestä suoraan muokkaamaan yksityiskohtia. Hierarkiarakenne haluttiin visualisoida kolmiulotteisena maailmana, laatikoista muodostuvina rakenteina, ja komponenttien yksityiskohdat, tekstimuotoiset dokumentit, haluttiin esittää kaksiulotteisen tekstieditorin sisällä. Siirtymä 3D:stä 2D:hen haluttiin tehdä sulavaksi.

Työtä voidaan pitää onnistuneena. Siirtymä 3D- ja 2D-objektien välillä saatiin sulavaksi käyttämällä siirtymässä semanttista zoomausta, sulavaa ja osittain automatisoitua liikettä. Alkuperäisessä suunnitelmassa haluttiin hallita zoomaustasoja laatikoiden läpinäkyvyyden avulla, mutta siitä jouduttiin luopumaan läpinäkyvyyden teknisen toiminnan takia. Tilalle keksittiin kuitenkin ratkaisu, jossa laatikot litistyvät zoomauksen yhteydessä paljastaen sisältönsä. Interaktiota objektien kanssa korostetaan

animaatioilla, mutta toiminta on osittain puuttellinen eri toimintojen yhdistyessä liikutetun klikkauksen aikana.

Tekstidokumentit saatiin esitettyä näkymässä kaksiulotteisina mainostaulutekniikan avulla. Tekstidokumenttien tilalle mietittiin alunperin näkymiä oikeista ohjelmistokehitysympäristöistä, kuten Visual Studiosta, mutta toteutuksessa käytetyn Qt 3D:n rajoitteista johtuen tyydyttiin tekstieditorin toteutukseen, joka riittää yksinkertaiseen tekstin muokkaukseen.

Prototyyppi vaatii vielä paljon jatkokehitystä, sillä sen nykyinen toiminta ei riitä varsinaiseen ohjelmiston kehitykseen, mutta riittää demonstrointitarkoitukseen. Järjestelmän ensisijainen sovelluskohde on Kactus2:ssa IP-XACT-komponenttien VLNV-hierarkiatasojen kuvaus ja sen sisältämien komponenttien muokkaus. Sen ohella vastaava järjestelmä soveltuisi myös muille ohjelmistotuotannon alueille. Yksinkertaisimmillaan sillä voi kuvata ohjelmaprojektin kansiorakenteita eri ohjelmointitympäristöissä. Se on sovellettavissa myös yleisesti ohjelma-arkkitehtuurien kuvaukseen, etenkin jos komponenttien välillä voitaisiin ilmaista hierarkiatasojen lisäksi myös muita riippuvuussuhteita. Kun komponenttien yhteydessä pystytään vielä näyttämään komponentin lähdekoodia oikeissa ohjelmistokehityksen työkaluissa, ovat käyttömahdolliset laajat.

LÄHTEET

[1] P. Caserta, O. Zendra, Visualization of the Static Aspects of Software: A Survey, IEEE Transactions on Visualization and Computer Graphics, Vol. 17, Iss. 7, 2011, pp. 913-933. [Viitattu 30.4.2018]. Saatavissa: https://ieeexplore.ieee.org/document/5557869.

[2] A.R. Teyseyre, M.R. Campo, An Overview of 3D Software Visualization, IEEE Transactions on Visualization and Computer Graphics, Vol. 15, Iss. 1, 2009, pp. 87-105. [Viitattu 30.4.2018]. Saatavissa: https://ieeexplore.ieee.org/document/4564449.

[3] T.D. Hamalainen, E. Salminen, Gamification of System-on-Chip design, 2014 International Symposium on System-on-Chip (SoC), IEEE, pp. 913-933. [Viitattu 30.4.2018]. Saatavissa: https://doi.org/10.1109/ISSOC.2014.6972441.

[4] Unity3D. User Interfaces for VR. [Viitattu 7.5.2018]. Saatavissa: https://unity3d.com/learn/tutorials/topics/virtual-reality/user-interfaces-vr.

[5] Unreal Engine, Creating 3D Widget Interaction. [Viitattu 7.5.2018]. Saatavissa: https://docs.unrealengine.com/en-us/Engine/UMG/HowTo/InWorldWidgetInteraction.

[6] D. Ignacio, Crafting Destruction: The Evolution of the Dead Space User Interface, Game Developers Conference 2013. [Viitattu 7.5.2018]. Saatavissa: http://www.gdcvault.com/play/1017723/Crafting-Destruction-The-Evolution-of.

[7] Qt Blog. Widgets enter the third dimension: WolfenQt. [Viitattu 2.4.2018]. Saatavissa: http://blog.qt.io/blog/2008/12/02/widgets-enter-the-third-dimensionwolfenqt/.

[8] B. Shneiderman, The eyes have it: a task by data type taxonomy for information visualizations, Proceedings 1996 IEEE Symposium on Visual Languages, pp. 336-343. [Viitattu 30.4.2018]. Saatavissa: https://ieeexplore.ieee.org/document/545307/.

[9] Google. Google Maps. [Viitattu 10.5.2018]. Saatavissa: https://www.google.com/maps.

[10] A. Kamppi, E. Pekkarinen, J. Virtanen, J-M. Määttä, J. Järvinen, L. Matilainen, M. Teuho, T. D. Hämäläinen, Kactus2: A graphical EDA tool built on the

IPXACT standard, The Journal of Open Source Software, 2017, pp. 151. [Viitattu 30.4.2018]. Saatavissa: http://joss.theoj.org/papers/10.21105/joss.00151

[11] OpenGL. [Viitattu 10.5.2018]. Saatavilla: https://www.opengl.org/.

[12] The Qt Company Ltd. Qt 3D Overview. [Viitattu 19.2.2018]. Saatavissa: https://doc.qt.io/qt-5.9/qt3d-overview.html.

[13] The Qt Company Ltd. Signals & Slots. [Viitattu 18.3.2018]. Saatavissa: http://doc.qt.io/qt-5.9/signalsandslots.html.

[14] The Qt Company Ltd. Qt 3D Render Framegraph. [Viitattu 19.2.2018]. Saatavissa: https://doc.qt.io/qt-5.9/qt3drender-framegraph.html.

[15] The Qt Company Ltd. QForwardRenderer. [Viitattu 2.4.2018]. Saatavissa: https://doc.qt.io/qt-5.9/qt3dextras-qforwardrenderer.html.

[16] The Qt Company Ltd. QRenderSettings. [Viitattu 27.4.2018]. Saatavissa: https://doc.qt.io/qt-5.9/qt3drender-qrendersettings.html.

[17] The Qt Company Ltd. QML Applications. [Viitattu 11.5.2018]. Saatavissa: https://doc.qt.io/qt-5.9/qmlapplications.html.

[18] T. McReynolds, B. David, Advanced Graphics Programming Using OpenGL, Morgan Kaufmann Publishers, San Francisco, 2005, 644s.

[19] The Qt Company Ltd., Klaralvdalens Datakonsult AB. renderer.cpp. [Viitattu 28.2.2018]. Saatavissa: http://code.qt.io/cgit/qt/qt3d.git/tree/src/render/backend/renderer.cpp?h=5.9.

[20] Klaralvdalens Datakonsult AB. qphongalphamaterial.cpp. [Viitattu 18.3.2018]. Saatavissa: http://code.qt.io/cgit/qt/qt3d.git/tree/src/extras/defaults/qphongalphamaterial.cpp?h=5.9.

[21] A. Puhakka, 3D-grafiikka, Talentum Media, Helsinki, 2008, 453s.

[22] The Qt Company Ltd. Qt Quick Controls 2 - Text Editor. [Viitattu 7.5.2018]. Saatavissa: http://doc.qt.io/qt-5.9/qtquickcontrols2-texteditor-example.html.